Journal of Machine Learning **ISSN**: 2790-2048(e), 2790-203X(p)

Stochastic Operator Network: A Stochastic Maximum Principle Based Approach to Operator Learning

Ryan Bausback[1], Jingqiao Tang[1], Lu Lu[2], Feng Bao[1], and Phuoc-Toan Huynh* [1]

[1]Department of Mathematics, Florida State University, Florida 32304, Tallahassee, USA.
[2]Department of Statistics and Data Science, Yale University, New Haven, CT 06511, USA.

Abstract. We develop a novel framework for uncertainty quantification in operator learning, namely the stochastic operator network (SON). SON combines the stochastic optimal control concepts of the stochastic neural network (SNN) with the deep operator network. By formulating the branch net as an SDE and back-propagating through the adjoint BSDE, we replace the gradient of the loss function with the gradient of the Hamiltonian from the stochastic maximum principle in the stochastic gradient descent (SGD) update. This allows SON to learn the uncertainty present in operators through its diffusion parameters. We then demonstrate the effectiveness of SON when replicating several noisy operators in 2D and 3D.

Keywords:
Stochastic operator learning,
Deep operator network,
Stochastic maximum principle,
Backward stochastic differential equation,
Uncertainty quantification.

Article Info.:
Volume: X
Number: X
Pages: 1 - 26
Date: /2026
doi.org/10.4208/jml.250709

Article History:
Received: 09/07/2025
Accepted: 16/12/2025

Communicated by:
Zhi-Qin Xu

1 Introduction

Over the past five years, operator learning has emerged as a promising alternative to traditional numerical solvers for a wide range of differential equations [5, 9, 14, 18, 21, 22, 26, 30, 38]. Unlike classical machine learning models that approximate functions at discrete spatial or temporal points, neural operators take input as an entire function and produce a corresponding output function [25]. The two most influential architectures, deep operator network (DeepONet) and Fourier neural operator (FNO), have driven the evolution of deep operator learning. DeepONet, which is based on the universal approximation theorem for operators [9], comprises two parallel subnetworks: the "branch" net, which learns coefficients, and the "trunk" net, which learns a data-driven basis for the output function. FNO, in contrast, encodes and decodes using the Fourier basis (via fast Fourier transform (FFT) and inverse FFT) within successive Fourier layers, relying on a single feed-forward backbone and assuming that input and output domains coincide [22]. The authors in [26] later showed that FNO was simply a special parameterization of DeepONet under the Fourier basis, which means that all variants of the FNO benefits from the same universal approximation feature as the DeepONet [26]. Crucially, neither DeepONet nor FNO relies on a fixed mesh discretization, a key limitation of many classical PDE solvers in solving large scale problems, which paves the way for rapid, mesh-free inference across a broad class of parameterized PDEs.

*Corresponding author. th24ba@fsu.edu.

While DeepONet has been applied to certain stochastic problems, such as stochastic ordinary differential equations (ODEs) and stochastic PDEs in [27], the randomness appears only in the input function, not as intrinsic stochasticity in the operator itself. The network still learns a deterministic map from noisy inputs to outputs, rather than an operator that injects noise at each point along a trajectory. Beyond these stochastic input applications, several advanced operator learning frameworks have emerged. For example, GANO (generative adversarial neural operator) uses an FNO backbone to train a generator-discriminator pair for richer output sampling, and PCAnet replaces the Fourier basis with a data-driven principal component analysis (PCA) basis to reduce dimensionality [5, 30]. A handful of architectures explicitly target uncertainty quantification, for example, the information bottleneck-uncertainty quantification (IB-UQ) replaces the branch network with an encoder-decoder that learns a latent representation of input noise to produce predictive distributions [14]. However, most operator-learning work still focuses on deterministic accuracy, and even UQ-oriented variants have not yet tackled SDE-style operators with pointwise noise along trajectories.

In this work, we propose a novel training strategy for stochastic operators by merging probabilistic learning, namely, stochastic neural networks, with the DeepONet framework. The SNN is an extension of the so-called "neural ODE" network architecture where the evolution of hidden layers in the deep neural network (DNN) is formulated as a discretized ordinary differential equation system [8, 10, 13, 15, 34]. More specifically, an additive Brownian motion noise, which characterizes the randomness caused by the uncertainty of models and noises of data, is added to the ODE system corresponding to the hidden layers to transform the ODE into an stochastic differential equation (SDE) [16, 17, 23, 24, 33]. In the SNN model, the prediction of the network is represented through the drift parameters, and the stochastic diffusion governs the randomness of network output, which serves to quantify the uncertainty of deep learning. Compared to other probabilistic learning methods, such as the Bayesian neural network (BNN) [11, 12, 19, 29, 31, 35–37], the SNN approach takes less computational cost to evaluate the diffusion coefficient, while it is still able to characterize sufficient probabilistic behavior of the neural network by stacking (controlled) diffusion terms together through the multilayer structure [2]. The main challenge in implementing the SNN is to construct an efficient numerical solver for the backpropagation process, since the standard backpropagation approach used in the neural ODE is deemed computationally inefficient [2, 3]. We address this by adopting the sample-wise backward-SDE solver of [2, 3], which treats backward samples as "pseudo data" and solves only a small, randomly selected subset of the backward SDE per iteration, thus drastically reducing computational burden.

The main contributions of our paper are as follows. We first construct the general framework of our training strategy, namely the stochastic operator network, which effectively combines the methodology of the DeepONet and the training process of the SNN. The stochastic maximum principle (SMP) [8] is adopted to formulate the loss function of the new backpropagation process and transforms the training algorithm of the neural operator into a stochastic optimal control problem. We then consider several numerical experiments to validate our method on a range of stochastic operators. Finally, we also compare the performance our method with the standard DeepONet approach.

The rest of this paper is organized as follows. We will detail the intricacies of both methods forming the foundation of SON, elaborating on DeepONet in Section 2, and SNN in Section 3. In Section 4, we introduce the stochastic operator network and discuss the details of how SNN and DeepONet work in tandem to produce SON. Finally in Section 5 we prove the effectiveness of this new architecture by applying it to several noisy 2D and 3D operators, with similar outputs to stochastic differential equations. We also compare SON performance with that of the vanilla DeepONet.

2 Deep operator network

Deep operator network is among the most well-known operator learning architectures, whose core concept is to provide an estimation to a given mapping between two Banach spaces [18]. Given an operator G taking an input function u from some input space V, DeepONet returns an approximation operator for $G(u)$ over the set of points y in the domain of $G(u)$. Hence, the network takes inputs composed of two parts: u and y, and outputs $G(u)(y)$ [27]. A simple illustration of the architecture of DeepONet can be found in Fig. 2.1. To enable the training procedure of the approximation network, we discretize the input function u by sampling its values at a finite set of m locations $\{x_1, \ldots, x_m\}$ called "sensors". Our discretization for any u is therefore $\{u(x_0), \ldots, u(x_m)\}$. Although various representations are possible (see, e.g., [27]), we adopt this simplest approach unless stated otherwise. In our experiments, we use the same m sensors for every input, but more flexible schemes where sensor locations vary by sample have also been explored (see, e.g., [38]).

The theoretical foundation of operator learning is the following universal approximation theorem for operators, first discovered by the authors in [9].

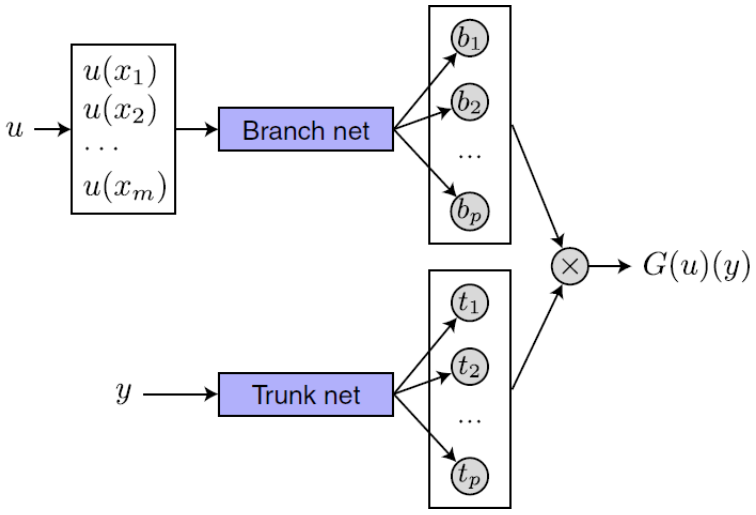

Figure 2.1: Vanilla DeepONet architecture [25]. The branch networks learns input function $u(x)$ evaluated at m sensors, while the trunk net learns the domain y of operator output function $G(u)(y)$.

Theorem 2.1 (Universal Approximation Theorem for Operators, [9]). *Suppose σ is continuous non-polynomial function, X is Banach space, $K_1 \subset X, K_2 \subset \mathbb{R}^d$ are compact sets, V is compact in $C(K_1)$, and G is operator mapping V to $C(K_2)$. Then for $\epsilon > 0$, there are positive integers n, p and m, constants c_i^k, ξ_{ij}^k, $\theta_i^k \in \mathbb{R}$, $w_k \in \mathbb{R}^d$, $x_j \in K_1$, $i = 1, \ldots, n$, $k = 1, \ldots, p$ and $j = 1, \ldots, m$ such that*

$$\left| G(u)(y) - \sum_{k=1}^{p} \underbrace{\sum_{i=1}^{n} c_i^k \sigma \left(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k \right)}_{Branch} \underbrace{\sigma(w_k y + \zeta_k)}_{Trunk} \right| < \epsilon$$

holds for all $u \in V$ and $y \in K_2$. Here, $C(K)$ is the Banach space of all continuous functions defined on K with norm $\|f\|_{C(K)} = \max_{x \in K} |f(x)|$.

While this result ensures that a shallow branch-trunk pair can in principle approximate any continuous operator, it restricts both subnetworks to have a single hidden layer and identical structure. Lu *et al.* [27] removed these constraints and proved an analogous approximation guarantee for arbitrary (potentially deep and asymmetric) branch and trunk architectures. We summarize their extension as Theorem 2.2 below.

Theorem 2.2 (Generalized Universal Approximation Theorem for Operators, [27]). *Suppose X is Banach space, $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ are two compact sets, respectively, V is compact in $C(K_1)$, and $G : V \to C(K_2)$ is a nonlinear continuous operator. Then, for any $\epsilon > 0$, there exist positive integers m, p, continuous vector functions $\boldsymbol{g} : \mathbb{R}^m \to \mathbb{R}^p$, $\boldsymbol{f} : \mathbb{R} \to \mathbb{R}^p$, and $x_1, \ldots, x_m \in K_1$ such that*

$$\left| G(u)(y) - \left\langle \underbrace{\boldsymbol{g}(u(x_1), u(x_2), \ldots, u(x_m))}_{Branch}, \underbrace{\boldsymbol{f}(y)}_{Trunk} \right\rangle \right| < \epsilon$$

holds for all $u \in V$ and $y \in K_2$, where $\langle \cdot, \cdot \rangle$ denotes the dot product in $\mathbb{R}^p$. Furthermore, the functions $\boldsymbol{g}$ and $\boldsymbol{f}$ can be chosen as diverse classes of neural networks, which satisfy the classical universal approximation theorem of functions, for examples, (stacked/unstacked) fully connected neural networks, residual neural networks and convolutional neural networks.

The generalization in Theorem 2.2 lets the branch and trunk networks differ in depth and width, provided they both output a p-dimensional vector. Assume that the output of the branch and the trunk networks are the vectors $[\beta_0(\boldsymbol{u}), \ldots, \beta_p(\boldsymbol{u})]^\top$, $[\tau_0(y), \ldots, \tau_p(y)]^\top$ respectively, where $\boldsymbol{u} = \{u(x_i)\}_{i=1}^m$. Then, the output of the DeepONet, the approximate operator of $G(u)$, is defined as

$$\hat{G}(u)(y) = \sum_{k=1}^{p} \beta_k(\boldsymbol{u}) \tau_k(y). \tag{2.1}$$

Alternatively, one can add bias to each β_k of the brand network and to the last stage of the network. Even though bias is not required in the statement of Theorem 2.2, adding bias may increase the performance by reducing the generalization error [27]. As a result, one

can replace the approximate operator $\hat{G}(u)$ in (2.1) by

$$\tilde{G}(u)(y) = \sum_{k=1}^{p} \tilde{\beta}_k(\boldsymbol{u})\tau_k(y) + b_0, \tag{2.2}$$

where $\tilde{\beta}_k = \beta_k + b_k$, $k = 1, \ldots, p$ and $\{b_k\}_0^p$ is the set of bias.

In operator-learning applications, we assemble the DeepONet training and testing data by forming the Cartesian product of two discrete input sets for the branch network and the trunk network. More specifically, for the branch inputs, a collection of n input functions $\{u_i\}_{i=1}^n$, each sampled at m fixed sensor locations $\{x_j\}_{j=1}^m$, which results in the feature vectors

$$\boldsymbol{u}_i = [u_i(x_1), \ldots, u_i(x_m)]^\top \in \mathbb{R}^m, \quad i = 1, \ldots, n.$$

Regarding the inputs for the trunk architecture, we consider a set of evaluation points $\{y_k\}_{k=1}^d$ in the operator output domain. By pairing each branch vector $\boldsymbol{u}_i$ with each evaluation point y_k, we obtain a total of $n \times d$ samples

$$(\boldsymbol{u}_i,\ y_k) \ \mapsto \ G(u_i)(y_k), \quad i = 1, \ldots, n, \quad k = 1, \ldots, d.$$

This construction ensures flexibility, as the sensor grid $\{x_j\}$ fed into the branch net and the evaluation points $\{y_k\}$ used by the trunk net may differ in both dimensionality and resolution, which allows diverse sampling strategies in high-dimensional operator learning tasks.

Despite their capabilities in learning the operators between spaces, vanilla DeepONet yields fully deterministic outputs and cannot capture the intrinsic noise inherent in solutions of stochastic differential equations. In the next section, we give a concise overview of the stochastic neural networks, which is a probabilistic operator learning framework that integrates an SDE-based diffusion term to recover the randomness occurring in stochastic operators.

3 Stochastic Neural Network

In this section, we review the mathematical formulation of stochastic neural networks introduced in [3], and summarize the sample-wise backpropagation algorithm for training them from [2]. As an extension of the neural-ODE architecture, the class of SNNs in [3] was formulated by stochastic differential equations, which describe stochastic forward propagation of deep neural networks. The training procedure of the SNNs is then designed as a stochastic optimal control problem, which is solved via a generalized stochastic gradient descent algorithm.

3.1 Mathematical formulation

The SNN structure that we consider in this work is given by

$$A_{n+1} = A_n + h\mu(A_n, \theta_n) + \sqrt{h}\sigma(\theta_n)\omega_n, \quad n = 0, 1, \ldots, N-1, \tag{3.1}$$

where

$$A_n := [a^1, a^2, \dots, a^L] \in \mathbb{R}^{L\times d}$$

denotes the vector that contains all the L neurons at the n-th layer in the DNN, h is a fixed positive constant that stabilizes the network, θ_n represents the trainable neural network parameters, such as weights and biases, that determine the output of the neural network, $\{\omega_n\}_n := \{\omega_n\}_{n=0}^{N-1}$ is a sequence of i.i.d. standard Gaussian random variables, σ is a coefficient function that determines the size of the uncertainty in the DNN, and μ is the pre-chosen activation function. We point out that the terms $\{\sigma(\theta_n)\omega_n\}$ allow the SNN to produce random output reflecting the stochastic behaviors of the target model [2, 3]. Unlike Bayesian neural networks, which treat the weights θ of a standard DNN as random variables and use Bayesian inference to approximate their posterior distribution, the SNN framework introduces uncertainty by injecting a learned noise term directly into the network dynamics.

If we choose a positive constant T as a pseudo terminal time and let $N \to \infty$ or $h \to 0$ with $h = T/N$, the dynamics of the SNN (3.1) becomes the stochastic differential equation, which, in the integral form, is given by

$$A_T = A_0 + \int_0^T \mu(A_t, \theta_t)dt + \int_0^T \sigma(\theta_t)dW_t, \tag{3.2}$$

where

$$W := \{W_t\}_{0\le t\le T}$$

is a standard Brownian motion corresponding to the i.i.d. Gaussian random variable sequence $\{w_n\}_n$ in (3.1), $\int_0^T \sigma(\theta_t)dW_t$ is an Itô integral, and A_T corresponds to the output of the SNN.

In this work, we shall treat the training procedure for deep learning as a stochastic optimal control problem and the parameter θ_t in (3.2) as a control process. This approach is supported by the Stochastic Maximum Principle, and then translated into an implementable architecture with diverse applications [2]. An alternative formulation based on the dynamic programming principle (DPP) leads to Hamilton-Jacobi-Bellman (HJB) equations. However, the high dimensionality of neural network parameter spaces renders the direct solution of HJB equations computationally intractable. Consequently, most SNN architectures rely on SMP rather than DPP to ensure scalability and numerical tractability in learning stochastic operators.

To this end, let Γ be the random variable that generates the training data to be compared with A_T, we define the cost functional J for the optimal control problem as

$$J(\theta) = E\left[\Phi(A_T, \Gamma) + \int_0^T r(A_t, \theta_t)dt\right],$$

where

$$\Phi(A_T, \Gamma) := \|A_T - \Gamma\|_{\text{loss}}$$

is a loss function corresponding to a loss error norm $\|\cdot\|_{\text{loss}}$, and the integral $\int_0^T r(A_t, \theta_t)dt$ represents the running cost in a control problem. The goal of the deep learning is to solve

the stochastic optimal control problem, i.e., find the optimal control θ^* such that

$$J(\theta^*) = \inf_{\theta \in \Theta} J(\theta), \tag{3.3}$$

where Θ is an admissible control set.

We assume that the control space Θ is convex and that the control $\theta_t \in \Theta$ is an admissible control [4]. An admissible control is one that is both square integrable: $\int_{-\infty}^{\infty} |\theta_s|^2 ds < \infty$ and adapted to the filtration $\mathbb{F}$ generated by W_s. Since θ_t is the set of neural network parameters, we think of $\Theta = \mathbb{R}$ or that these weights and biases might become any real number, thus ensuring that Θ is convex.

The stochastic maximum principle allows us to solve the optimal control problem by minimizing the Hamiltonian defined by

$$H(a, b, c, \theta) = r(a, \theta) + b\mu(a, \theta) + c\sigma(\theta).$$

The statement of the SMP is summarized as follows.

Theorem 3.1 (Stochastic Maximum Principle, [1]). *Let $\theta \in \Theta$ such that $E[|\theta|^2] < \infty$ and A^* be the optimally controlled process with optimal control θ^*. Then there exists a pair of adjoint processes (B^*, C^*) subject to*

$$\begin{aligned} dA_t^* &= \mu(A_t^*, \theta_t)dt + \sigma(\theta_t)dW_t, \\ dB_t^* &= -\big(r_a(A_t^*, \theta_t^*) + \mu_a(A_t^*, \theta_t^*)^T B_t^* + \sigma_a(\theta_t^*)^T C_t^*\big)dt - C_t^* dW_t \end{aligned} \tag{3.4}$$

with

$$B_T^* = \Phi_a(A_T^*), \quad C_t^* = \sigma_t \nabla_a B_t^*.$$

Additionally,

$$H(A_t^*, B_t^*, C_t^*, \theta_t^*) = \inf_{\theta \in \Theta} H(A_t^*, B_t^*, C_t^*, \theta_t).$$

In order to determine the optimal controls θ, the stochastic maximum principle yields the following gradient process [28, 36]:

$$\nabla_\theta J(\theta)\big|_t = \mathbb{E}\big[\mu_\theta^\top(t) B_t + \sigma_\theta^\top(t) C_t + r_\theta(t)\big]. \tag{3.5}$$

When θ parametrizes a neural network, one may perform stochastic gradient descent by updating the SNN weights using the gradient of the Hamiltonian [4, 7]

$$\begin{aligned} \theta_t^{i+1} &= \mathcal{P}_\Theta\big(\theta_t^i + \alpha E[\nabla_\theta H(A_t, B_t, C_t, \theta)]\big) \\ &= \mathcal{P}_\Theta\big(\theta_t^i + \alpha\big[r_\theta\big(A_t^i; \theta_t^i\big) + \mu_\theta^\top\big(A_t^i; \theta^i\big) B_t^i + \sigma_\theta^\top\big(\theta_t^i\big) C_t^i\big]\big), \end{aligned} \tag{3.6}$$

where α is the learning rate and $\mathcal{P}_\Theta$ denotes the projection onto the set of admissible controls. The convergence of this SGD iteration under standard smoothness assumptions is established in [2]. Numerical implementation of (3.6) requires approximations of the forward-backward SDE system (3.4). To this end, we briefly review the numerical schemes used in this work.

3.2 Numerical approximation for the forward and backward SDEs

To implement the SNN SGD iteration, we discretize the pseudo-time interval $[0,1]$ with a uniform partition

$$\Pi_N = \{t_n : 0 = t_0 < t_1 < \cdots < t_N = 1\},$$

where N denotes the number of SNN layers and $h = 1/N$ is the step size. Following [4], the forward SDE for A_t is approximated by the Euler-Maruyama scheme

$$A_{t_{n+1}} = A_{t_n} + \mu(A_{t_n}, \theta_{t_n})\, h + \sigma(\theta_{t_n})\, \Delta W_n, \quad \Delta W_n = \sqrt{h}\, \epsilon_{t_n}, \quad \epsilon_{t_n} \sim \mathcal{N}(0,1),$$

which acts as the forward pass of the stochastic neural network.

For the adjoint processes (B_t, C_t), the main difficulty is that the backward SDE evolves from t_{n+1} to t_n while B_t and C_t must remain adapted to the filtration

$$\mathcal{F}_t^W := \sigma(W_s, 0 \le s \le t).$$

Thus, when computing B_{t_n} from $B_{t_{n+1}}$, we only have information up to time t_n and must work with conditional expectations with respect to $\mathcal{F}_{t_n}^W$. To obtain the discretized equations for B_t and C_t, we use conditional expectations together with the Itô isometry and then replace these conditional expectations by a single Monte Carlo sample, yielding the following system (see, e.g., [4, 6]):

$$\begin{aligned} B_{t_n} &= B_{t_{n+1}} + h\, \nabla_a H(A_{t_{n+1}}, B_{t_{n+1}}, C_{t_n}, \theta_{t_n}), \\ C_{t_n} &= \frac{B_{t_{n+1}} \Delta W_n}{h} = \frac{B_{t_{n+1}} \epsilon_{t_n}}{\sqrt{h}}, \qquad\qquad n = N-1, \ldots, 0. \end{aligned} \tag{3.7}$$

These updates form the first step of the backward pass in training SNN and are used to compute the Hamiltonian whose gradient drives the parameter update (3.6).

Finally, we integrate the SNN training procedure into the DeepONet architecture to obtain a unified stochastic operator-learning framework, which we name the stochastic operator network.

4 Implementation of stochastic operator network

Stochastic operator network replaces the branch network of DeepONet with the stochastic neural network, allowing it to learn noisy operators such as stochastic ODEs through uncertainty quantification. Although the specific drift and diffusion sub-networks in the Euler-SDE scheme vary with each application, SON consistently trains both sets of parameters to capture the pointwise mean and variance of the operator's output.

Suppose we want to approximate the following noisy operator $G(u)(y, \epsilon)$ where $u(x)$ is the input function and ϵ is a random noise for each y that obscures the operator output function in $\mathcal{Y}$. To solve this problem, we first define the forward neural SDE for N pseudo-time steps in $t \in [0, T]$ where $T = 1$,

$$A_{t_{n+1}} = A_{t_n} + \mu(A_{t_n}, \theta_{t_n})\Delta t + \sigma(A_{t_n}, \theta_{t_n})\sqrt{\Delta t}\epsilon, \quad n = 0, \ldots, N-1, \tag{4.1}$$

where $\Delta t = 1/N$, and $\mu_{t_n}(\cdot, \theta)$ is produced by a neural network with trainable parameters θ at each time t_n. Likewise, the diffusion coefficient $\sigma(\cdot, \theta)$ can either be implemented as a simple list of trainable parameters or, like the drift, be defined by its own neural network. The branch net is therefore defined as the final state of the SDE (4.1) after N pseudo-time steps

$$\beta(u; \theta_\beta, \epsilon) := A_{t_N}(u, \epsilon; \theta_\beta),$$

where $\theta_\beta = \{\theta_{t_n}\}_{n=0}^{N-1}$ collects all parameters from the drift term and the diffusion term.

Therefore, a single branch net layer of SON is one forward propagation of the Euler-Maruyama scheme by $1/N$ along the SNN pseudo time interval $[0, 1]$.

The trunk net is a standard feed-forward neural network that maps the coordinate y to a feature vector of the same dimension as $\beta(u; \theta_\beta, \epsilon)$. We denote its output by $\tau(y; \theta_\tau)$, where θ_τ collects all trunk-network parameters. The output of both the SNN and trunk net should be vectors in $\mathbb{R}^p$. As with DeepONet, SON combines the outputs of these parallel structures together with the inner product to produce the operator output

$$\hat{G}_{SON}(u)(y) = \langle \beta(u; \theta_\beta, \epsilon), \tau(y; \theta_\tau) \rangle + b_0, \tag{4.2}$$

where $\langle \cdot, \cdot \rangle$ is the standard dot product and b_0 is a trainable bias parameter to improve generalization.

In our framework, the uncertainty inherent to the target stochastic operator is generated during the forward pass: we encode the input function $u(x)$ into an initial latent state and then evolve that state under an SDE whose drift and diffusion networks both depend on u. The resulting terminal state of this SDE serves as the output for the branch network, embedding randomness into the operator mapping itself rather than adding noise to an otherwise deterministic output.

Although we integrate DeepONet's reconstruction layer onto an SNN ResNet backbone, we cannot simply train its parameters by standard backpropagation alone. Because the ResNet corresponds to a time-discretized SDE under stochastic-optimal-control, weight updates must follow the Hamiltonian-gradient conditions of the stochastic maximum principle defined in Section 3. In practice, SON performs one outer DeepONet evaluation per SNN step and then applies the adjoint-BSDE update to the drift and diffusion weights. Therefore, SON is more mechanically akin to adding DeepONet as the final layer of stochastic neural network. The full training loop is summarized in Algorithm 1.

It is important to note that the dimension of u during SNN forward propagation must remain consistent, so any projection of data must be done in layers before or after the Euler-Maruyama iteration [2]. If we have projections, then we project from the space $C(K_1)$ where our input functions $u(x)$ exist down to the space where we want to propagate the SDE, run the SDE forward, then project down to the space where the inner product takes place. Calculating B_t for projections while backpropagating is akin to setting $C_t = 0$, since no additional noise is added by the network during the forward pass. However, this is not the case for the final layer of the SON, where the inner product between the trunk and the SNN is taken. In this layer, the dimension of the state is no longer conserved. Nevertheless, it was shown in [32] that the architecture of the SON can be modified to handle layers whose output dimension changes (see the last paragraph of [32, Section 6.1.1, Page 96]).

Algorithm 1 SON Training Procedure.

For each epoch i in $i = 0, \ldots, I$:

1. Randomly draw a training pair (u, y) from the dataset, where the input function u is discretized at grid points $\{x_0, \ldots, x_m\}$ into the vector
$$\boldsymbol{u} = (u_0, \ldots, u_m) = \big(u(x_0), \ldots, u(x_m)\big).$$
2. Let $A^i_{t_0} = \boldsymbol{u}$. For $n = 0, \ldots, N-1$,
   (a) Compute $\mu(A^i_{t_n}; \theta^i)$ and $\sigma(A^i_{t_n}; \theta^i)$ using neural-network layers. The diffusion term $\sigma(A^i_{t_n}, \theta^i)$ can be replaced by a single trainable scalar σ^i.
   (b) $A^i_{t_{n+1}} \leftarrow A^i_{t_n} + \mu(A^i_{t_n}; \theta^i_{t_n})\Delta t + \sigma(A^i_{t_n}; \theta^i_{t_n})\sqrt{\Delta t}\epsilon$.
3. $\hat{y} \leftarrow \text{Trunk}(y)$.
4. $\hat{G}_{SON}(u)(y) \leftarrow \langle \hat{y}, A^i_{t_N} \rangle$
5. $B^i_{t_N} = \nabla_u \Phi(\hat{G}_{SON}(u)(y))$.
6. Solve backward for each $B^i_{t_n}$ and $C^i_{t_n}$ by computing Hamiltonian and taking $\nabla_u H(\cdot)$
7. $\theta^{i+1}_t = \theta^i_t + \alpha \nabla_\theta H(A^i_{t_n}, B^i_{t_n}, C^i_{t_n}; \theta)$.

More specifically, we treat the dot product between the branch network and the trunk network as an additional hidden layer in the SNN and then apply the modification in [32] to this final layer. The trainable parameters of the trunk network are now treated as additional parameters in the SNN and are also included in the Hamiltonian loss.

5 Numerical experiments

5.1 Data generation

To construct the train and test datasets, input functions $u(x)$ were sampled from a mean zero Gaussian random field (GRF) using the radial basis function (RBF) covariance kernel with length scale parameter $l = 0.2$, as in [25]. Values y were also randomly sampled from the operator output domain and then the Cartesian product of these two sets was formed. A single training example is therefore $(y, u(x))$ where $y \in \mathbb{R}^n$, $u(x) \in \mathbb{R}^m$ is an array of input function values, n is the dimension of the domain of operator output $G(u)(y)$, and m is the number of sensors, x. The correct operator output values $G(u)(y)$ were then calculated numerically using explicit Runge Kutta (RK45) in JAX.

In this section, we carry out five numerical experiments to demonstrate the performance of our proposed method in learning the stochastic operators. We then compare

the results obtained by our method with those produced by the vanilla DeepONet. Although Bayesian neural networks and the IB-UQ framework are alternative approaches to uncertainty quantification, we do not include them in our numerical comparisons for two reasons. First, BNN-based methods are prohibitively expensive in our setting, where the effective number of training samples is the product of the number of input functions and sensor locations [14]. Second, as claimed by the authors in [14], IB-UQ is designed to provide pointwise uncertainty estimates for $G(u)(y)$ at a single output location y, whereas in this work we aim to quantify joint uncertainty over the entire field.

5.2 Antiderivative operator with noise

We first consider the following antiderivative operator with noise:

$$G : u(x) \quad \rightarrow \quad s = s(0) + \int_0^y u(\tau)d\tau + \alpha\epsilon$$

with the initial condition $s(0) = 0$ and $\epsilon \sim N(0, 1)$ noise scaled by constant $\alpha = 0.1$. We aim to train a neural network to approximate the target operator and provide an estimation to the scaling constant α.

One hundred training $u(x)$ functions were sampled from GRF, with 100 sensors uniformly partitioning $x \in [0, 5]$, and uniformly randomly sampled 100 training points t from the operator output domain $t \in [0, 5]$. This gives $100 \times 100 = 10,000$ training examples. For testing, this was 1000 $u(x) \times 1000$ $t = 1,000,000$ examples, with the t uniformly partitioning $[0, 5]$.

For the trunk net, a 2 layer dense architecture with ReLU activation was implemented. For the SNN, 6 layers were used (i.e. $\Delta n = 1/6$ on $[0, 1]$ pseudo time). The drift μ was a 3 layer dense neural network with ReLU activation for each pseudo time step, while diffusion σ was a single trainable parameter at each pseudo time step initialized via $N(0, 1)$ normal distribution. Since dimension must be preserved during SNN forward propagation, μ had 100 neurons per layer as each $u(x)$ had 100 sensors. This gives the following form for the SNN forward discretized SDE:

$$u_{n+1} = u_n + \mu(u_n; \theta)\Delta n + \sigma(\theta)\sqrt{\Delta n}\epsilon, \quad n = 0, \ldots, N-1,$$

where θ are trainable weights.

SON trained for 2000 epochs with a 0.001 learning rate using ADAM and without batches. Mean squared error was the loss function for the initial BSDE backpropagation. One key thing to note is that while the loss maintains a generally convergent trajectory during operator learning, it can spike wildly due to the architecture and complexity of the problem. Therefore, a scheduler was used, multiplying the learning rate by 0.9 every 500 epochs after 1000.

Fig. 5.1 shows the integrals of four random $u(x)$ functions sampled from the test set. Red indicates the true integral values, blue indicates the antiderivative operator values with $N(0, 1)$ noise scaled by 0.1 added randomly at each point, and orange indicates the SON predictions. As Fig. 5.1 shows, SON is able to capture both the mean and variation of the operator output with a high degree of accuracy. The final mean square error (MSE)

after training was approximately equal to 0.04, as seen in Fig. 5.2, well within the level of scaled noise added to the operator output. Since the Hamiltonian loss is not restricted to be positive, the plot only shows instances where this loss is nonnegative under the log scaling.

In order to assess that SON has correctly quantified the uncertainty, every testing example was predicted by SON 100 times and the standard deviation calculated for each.

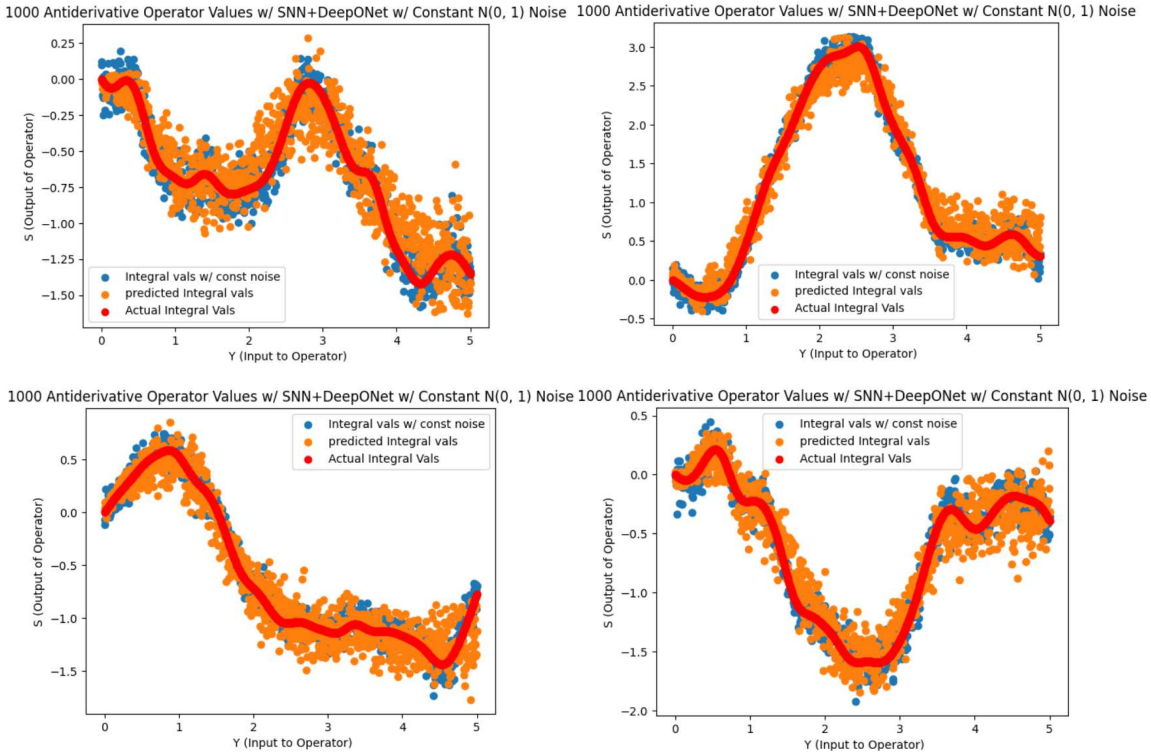

Figure 5.1: Antiderivative operator applied to four random test $u(x)$.

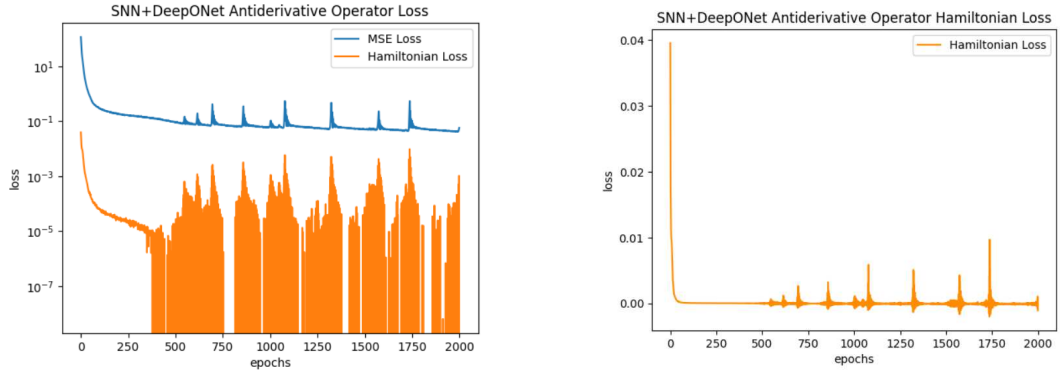

Figure 5.2: Noisy antiderivative operator loss over 2000 epochs.

Table 5.1: Average pointwise standard deviation for noisy antiderivative operator across multiple trainings.

	Trial 1	Trial 2	Trial 3
Recovered Noise	0.1430	0.1639	0.1354

This was then averaged over all testing examples (for both every u and every y). If SON is able to recover the scaling factor $\alpha = 0.1$, which was multiplied by the noise added to the data, then it has successfully quantified the uncertainty through the diffusion σ. As Table 5.1 shows, the recovered estimates are very close to the true scaling factor for three different trainings of SON on this problem.

5.3 ODE with noise

We next consider the perturbed solution operator to the following ODE:

$$\frac{ds(y)}{dy} = s(y)u(y)$$

with initial condition $s(0) = 1$. More specifically, the target stochastic operator takes the form

$$s(y) = s(0)e^{\int_0^y u(\tau)d\tau} + \alpha\epsilon = e^{\int_0^y u(\tau)d\tau} + \alpha\epsilon,$$

where $\epsilon \sim N(0,1)$ is the random noise scaled by $\alpha = 0.1$.

Training and testing data was generated as in Section 5.2, except our operator output domain was $t \in [0,1]$ to have a comparable range for all samples in the dataset. Keeping the domain $[0,5]$ resulted in difficulties learning the ODE's with much larger, outlier ranges. SON was also initialized as in Section 5.2.

As Fig. 5.3 shows, SON is able to once again able to approximate the ODE solution trajectory well for the four random test $u(x)$ shown. Final MSE was ≈ 0.0178 (Fig. 5.4) while SON was able to correctly quantify the uncertainty across two different trainings, as judged by the same standard deviation recovery test as before (Table 5.2).

5.4 2D ODE system with noise

Our third numerical experiment is the following nonlinear 2D system of ODEs:

$$\begin{aligned} \frac{ds_1}{dy} &= s_2, \\ \frac{ds_2}{dy} &= -\sin s_1 + u(y), \end{aligned} \tag{5.1}$$

where $s_0 = (s_1, s_2) = (0,0)$ and $y \in [0,1]$. We perturb the solution operator by adding a two-dimensional Gaussian noise vector

$$\boldsymbol{\varepsilon} = \alpha\tilde{\boldsymbol{\varepsilon}}, \quad \tilde{\boldsymbol{\varepsilon}} \sim N(0, \boldsymbol{I}_2), \quad \alpha = 0.1,$$

where $\tilde{\boldsymbol{\varepsilon}} = (\varepsilon_1, \varepsilon_2)$ with each component is sampled independently from $N(0,1)$.

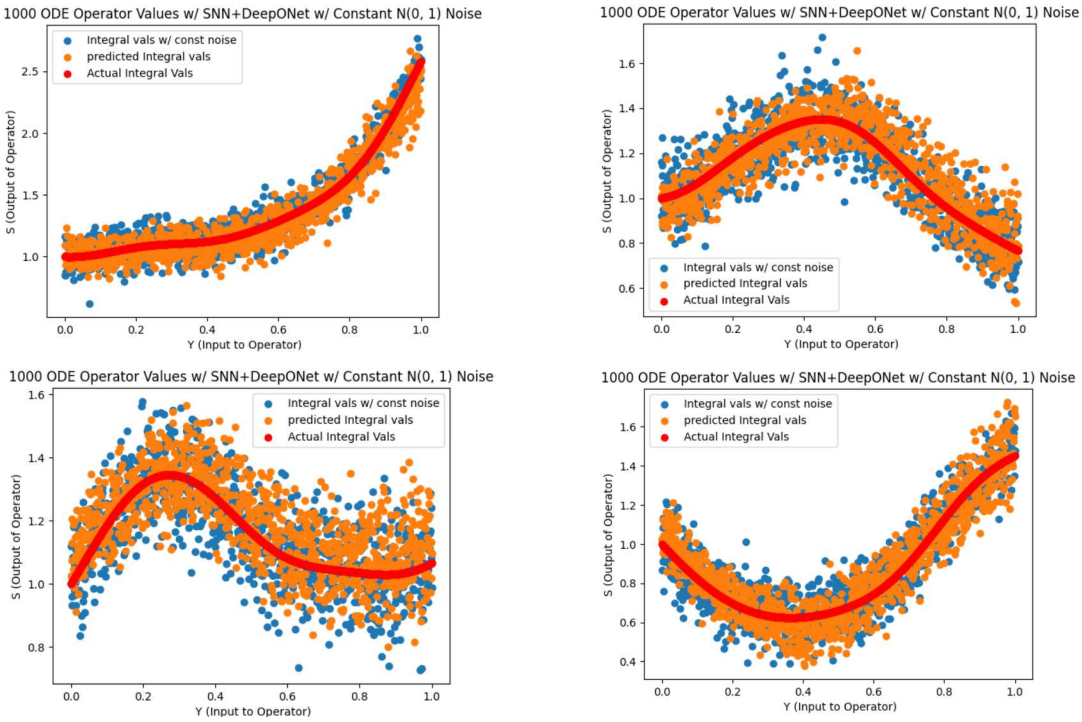

Figure 5.3: SON learning ODE applied to four random test $u(x)$.

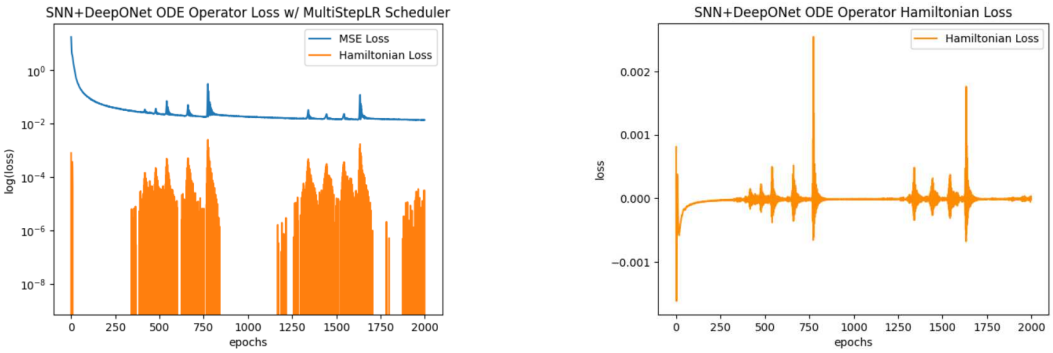

Figure 5.4: Noisy ODE operator loss over 2000 epochs.

Table 5.2: Average pointwise standard deviation for noisy ODE operator across multiple trainings.

	Trial 1	Trial 2	Trial 3
Recovered Noise	0.0925	0.0722	0.0985

As was done in [27], significant changes to SON needed to be implemented during the step where the SNN and trunk nets are combined in order to produce output with increased dimensions [27]. For this experiment, output of the trunk net was made twice

as large (200 neurons) as the SNN output (100 neurons) and then split in half to create two dimensional output. Therefore, in this instance, the first 100 neurons of the trunk net effectively provide information for s_1 in the domain, while the second 100 neurons provide information on s_2. These two pieces were then combined together with the same SNN output to produce the output of the noisy system $s = G(u)(y)$.

While most other hyperparameters and architectures remained the same as the previous experiments, the depth of SNN was increased to 10 layers and the diffusion parameters were initialized with $N(0, 2)$.

SON achieved a final MSE of 0.0162 after 2000 epochs, as seen in Fig. 5.5. Fig. 5.6 shows the pairs of predicted trajectories for three randomly sampled $u(x)$ in orange, as well the true trajectories (red) and with noise (blue). It is clear that SON is able to capture the true trajectory within the orange prediction band, as well as quantify the majority of noise on both dimensions. Additionally, after performing the same standard deviation test, SON recovered 0.069 and 0.17 as the noise level estimations for dimensions 1 and 2, which are both closely surrounding the correct scaling factor of $\alpha = 0.1$. This averages to an estimated noise quantification estimate of 0.12 averaged on both dimensions, or the noise scaling in the 2D spatial domain.

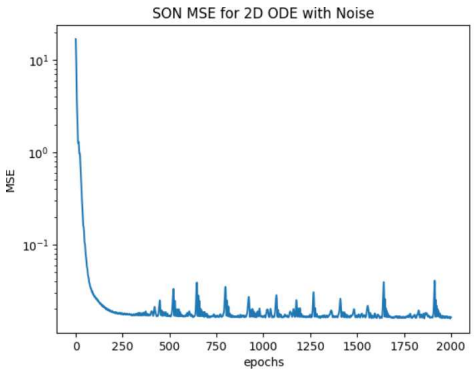

Figure 5.5: MSE loss for SON learning 2D ODE with noise.

5.5 Double integral with noise

We next consider the following double integral with noise:

$$s : u(\boldsymbol{x}) \;\; \rightarrow \;\; G(u)(\boldsymbol{y}) = \int_0^{y_1} \int_0^{y_2} u(\tau_1, \tau_2) d\tau_1 d\tau_2 + \alpha\epsilon,$$

where $\boldsymbol{x} \in \mathbb{R}^2, \boldsymbol{y} = (y_1, y_2) \in \mathbb{R}^2$ and $\epsilon \sim N(0, 1)$ noised scaled by $\alpha = 0.05$. Our map is from a space of 3D input $u(\boldsymbol{x})$ functions to a space of 3D operator output functions s. $u(\boldsymbol{x})$ is once again sampled from a mean-zero GRF with the RBF kernel with $l = 0.2$. However, $[0.5, 1.5]$ was now partitioned into 20 sensors along each dimension, resulting in 400 total sensors for each $u(\boldsymbol{x})$. The training and test sets had 100 and 20 $u(\boldsymbol{x})$ each respectively, which is close to the 80%/20% train/test split commonly used. The first $u(\boldsymbol{x})$ example in the training set is shown in Fig. 5.7.

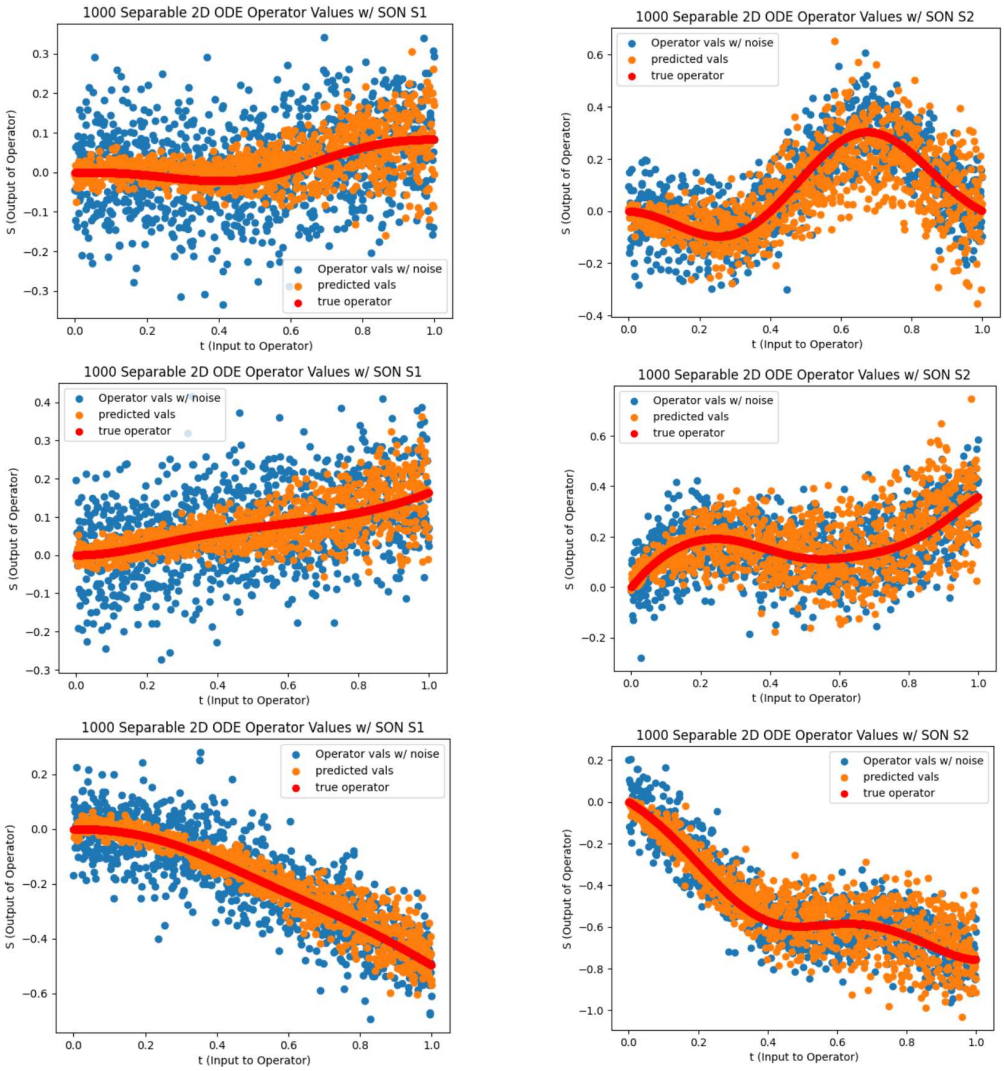

Figure 5.6: Pairs of SON predicted 2D ODE trajectories with 3 random test $u(x)$.

The operator output domain is now $\boldsymbol{y} \in [0.5, 1.5] \times [0.5, 1.5]$ to avoid an infinite values or discontinuities near 0. This domain was partitioned to give 30 input values along each dimension, resulting in $30 \times 30 = 900$ $\boldsymbol{y}$ for both test and training (Fig. 5.8).

The Cartesian product of the $u(\boldsymbol{y})$ and $\boldsymbol{y}$ was then taken and the double integral of each $(\boldsymbol{y}, u(\boldsymbol{x}))$ was computed numerically, resulting operator output surfaces as shown in the top left panel of Fig. 5.9. Noise was then added as described above (the top right panel of Fig. 5.9, where red is the true operator output and blue the noisy output used during SON training and testing).

As before, the trunk net was a 2 layer dense architecture, now with input dimension two instead of one and sigmoid activation. However, due to the size of the input data

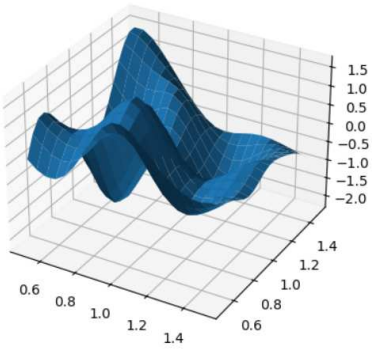

Figure 5.7: Example of $u(x)$ from training set.

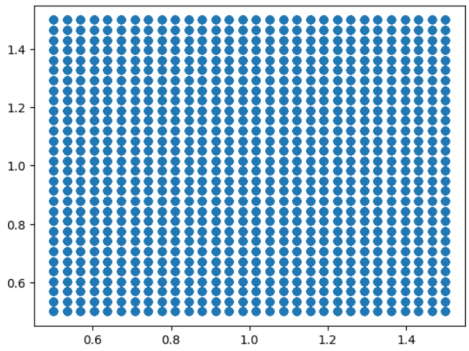

Figure 5.8: Operator output domain.

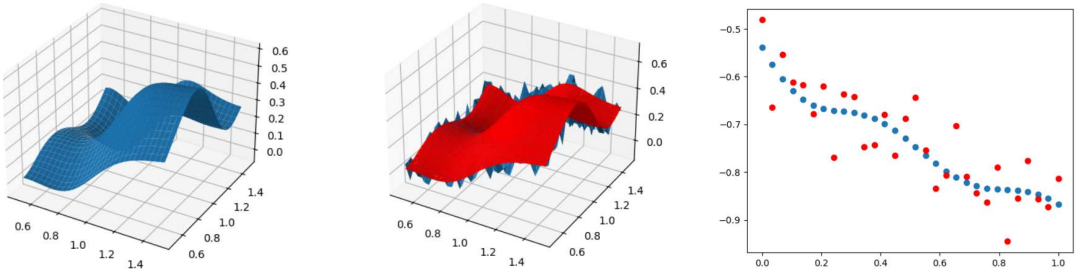

Figure 5.9: (Top left) Random output surface of double integral operator; (Top right) Double integral with scaled $N(0,1)$ noise (blue), true operator output (red); (Bottom) Cross section of operator output surface along $t_1 = 0.5$ axis.

to the SNN and its structure, the both the drift μ and diffusion σ were changed to stacks of convolutional layers. A single input to the SNN was hence 20×20 points from surface $u(x)$. For μ, a single layer with a kernel window size of 3 was employed with ReLU activation, while for σ, a single layer with kernel window size 3 and arctan activation was used, followed by a dropout layer with $p = 0.9$. This helped to prevent the predicted noise level from exploding after many training epochs. This gives the following form for the SNN forward discretized SDE:

$$u_{n+1} = u_n + \mu(u_n, \theta)\Delta n + \sigma(u_n, \theta)\sqrt{\Delta n}\epsilon, \quad n = 0, \ldots, N-1,$$

where θ are trainable weights. Five layers were used for the SNN, resulting in each pseudo-time-step $\Delta n = 1/5$.

Additionally, two projection layers were used in the SNN prior to the SDE propagation, where max pooling was utilized to further simplify the input function $u(x)$ and assist with surface feature extraction. The forward SDE output was then max-pooled a final time before being flatten for combination with the trunk net output.

SON was trained with ADAM for 200 epochs, with learning rate 0.001 reduced to 0.9 of the previous level by the scheduler every 25 epochs. The batch size was reduced to 900, meaning the double integrals of a single $u(x)$ were fed in per batch.

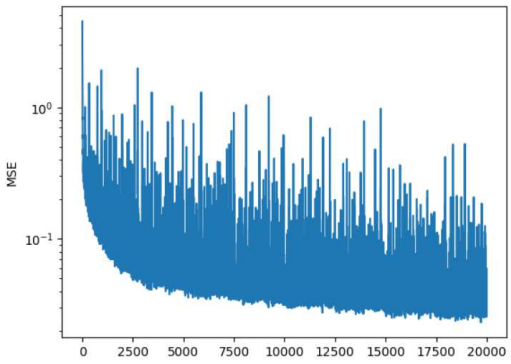

Figure 5.10: Training MSE loss for double integral operator with noise over 200 epochs and 100 batches per epoch.

Final MSE for the training set was 0.0296 (Fig. 5.10), with 0.043 MSE on the testing set. This indicates that SON is generalizing well to unknown data, and final MSEs are within the scaling factor of 0.05 of the added noise. However, although Fig. 5.10 does indicate convergence of the loss, it still fluctuates wildly despite utilization of scheduler to periodically reduce the learning rate. This is a common problem in operator learning, as even the initial DeepONet experiments in [20, 25].

As Fig. 5.11 shows for three random test set examples, SON is able to capture the mean underlying surface produced by the output of the double integral operator as well as quantify the uncertainty present.

Additionally, the standard deviation at all points of all operator outputs for all input functions $u(x)$ was calculated 20 times and the results averaged. This gave an estimate of 0.0359 on the noise level present, when then $N(0,1)$ noise was scaled by 0.05 before being added to the operator output. Given that these values are relatively close, this indicates that SON is able to correctly quantify the uncertainty present in this operator, confirming the intuition generated by the plots.

5.6 Stochastic elliptic equation with multiplicative noise

We adopt a test case from [27] and consider the solution operator of the following one-dimensional SPDE:

$$\operatorname{div}\left(e^{b(x;\omega)}\nabla u(x;\omega)\right) = f(x), \quad x \in (0,1], \quad \omega \in \Omega \tag{5.2}$$

with Dirichlet boundary conditions $u(0) = 0, u(1) = 1, f(x) = 5$, and

$$b(x;\omega) \sim \mathcal{GP}\left(b_0(x), \operatorname{Cov}(x_1, x_2)\right),$$

where the mean $b_0(x) = 0$ and the covariance function is

$$\operatorname{Cov}(t_1, t_2) = \sigma^2 \exp\left(-\frac{\|t_1 - t_2\|^2}{2l^2}\right)$$

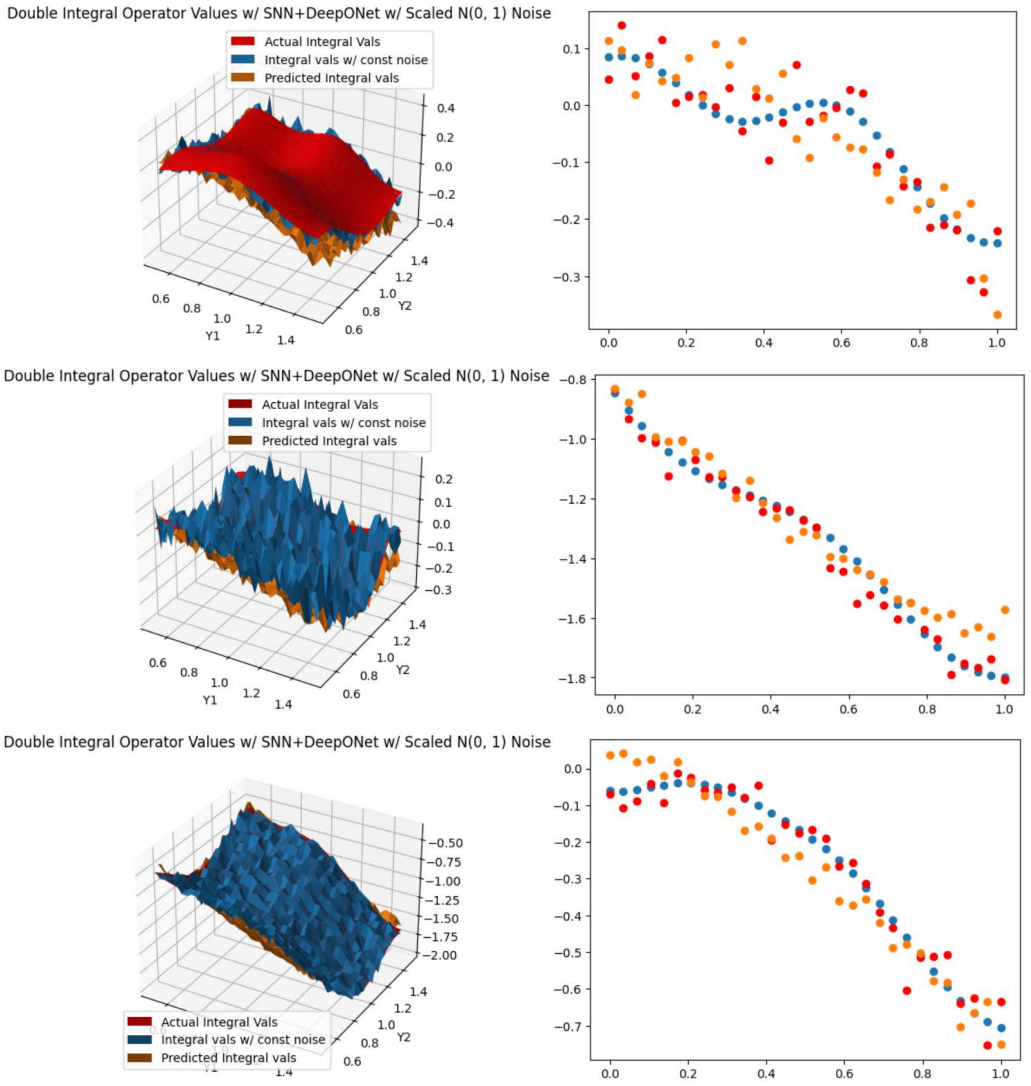

Figure 5.11: Three random SON double integral operator with noise test set predictions compared with true operator output (left) and corresponding random cross sections (right).

with $\sigma = 0.1$ and $l \in [1, 2]$. Unlike [27], we do not have to use Karhunen-Loève (KL) expansion for $k(t;\omega)$ to inject randomness into the trunk network. Instead, stochasticity is introduced automatically by propagating $k(t;\omega)$ through the SNN layers of the branch network. As a result, the input for the branch network is the random process $k(t\ \omega)$, while the trunk network only needs to take t as input. This significantly reduces the dimension of the training process. We train the SON with 5000 different $k(t;\omega)$ with l randomly sampled in $[1, 2]$ and for each $k(t;\omega)$, we only use one realization. The prediction for 8 different random samples from $k(t;\omega)$ with $l = 1.5$ and $l = 1.7$ are shown in Fig. 5.12. To further quantify the uncertainty, we generate 1000 samples and generate the prediction band in

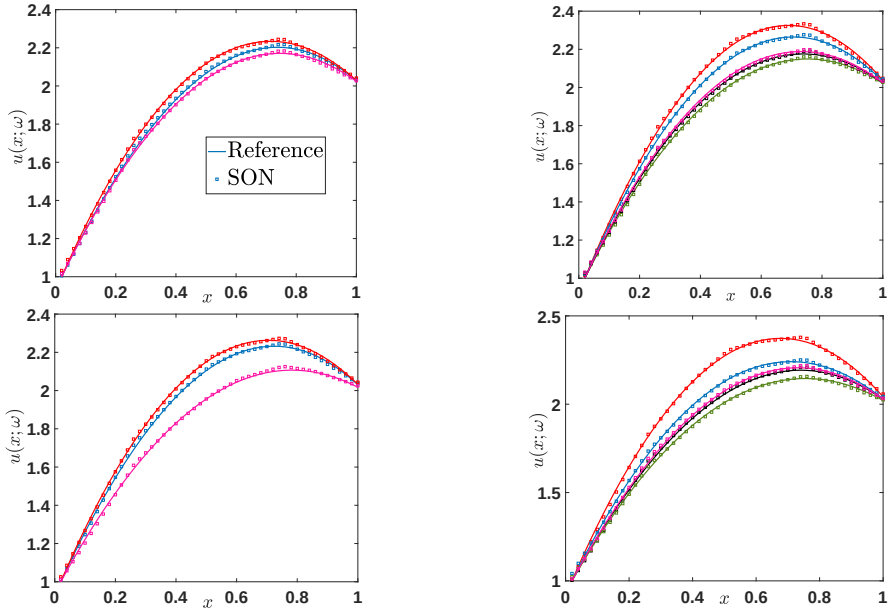

Figure 5.12: The prediction for 8 different random samples from $b(x;\omega)$ with (Top) $l = 1.5$. (Bottom) $l = 1.7$.

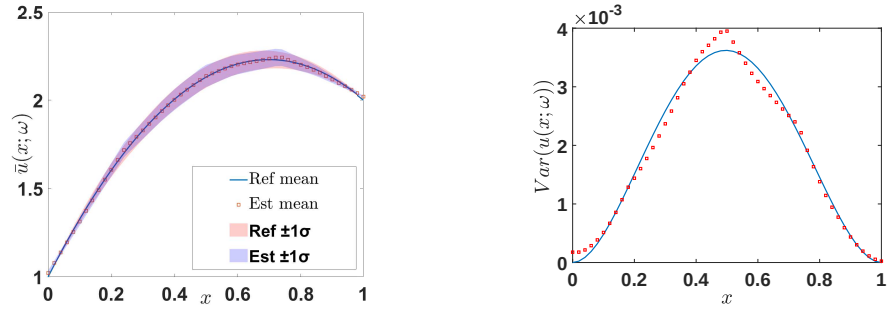

Figure 5.13: With $l = 1.5$ (Left) Mean and confidence band. (Right) Spread of the data.

Fig. 5.13 to visualize the spread of the data. The mean and variance of these samples are also shown in Fig. 5.13. Finally, the corresponding covariances and their differences are shown in Fig. 5.14.

It can be observed from the above figures that the estimated solution operator can reflect well the spread of the solution ensemble, as evidenced by its ability to match the sample variance and covariance of the reference data.

5.7 Geometric Brownian motion with time-dependent drift

Our final numerical test case is the following geometric Brownian motion with time-dependent drift term given by:

$$dS_t = \mu u_t S_t dt + \sigma S_t dW_t, \quad 0 < t \leq 1 \tag{5.3}$$

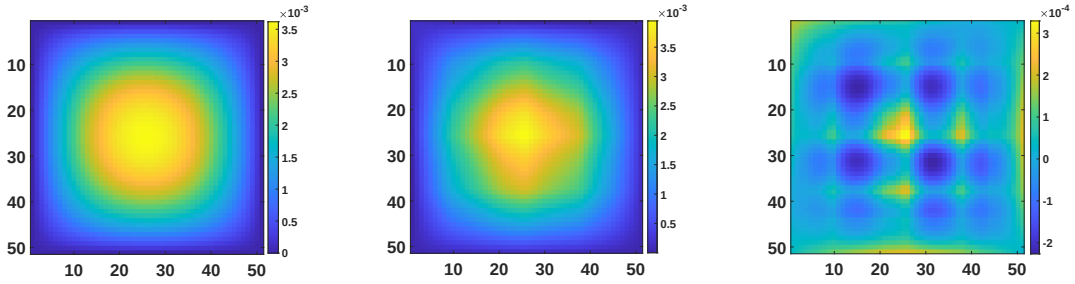

Figure 5.14: Covariance with $l = 1.5$: (Left) Reference. (Middle) Estimation. (Right) Difference.

with initial condition $S_0 = 1$ where u_t is the input function. The solution to Eq. (5.3) take the following exact formula:

$$S_t = S_0 \exp\left(\mu \int_0^t u(s)ds - \frac{\sigma^2}{2}t + \sigma W_t\right). \tag{5.4}$$

To generate the set of reference solutions for SON, we first divide $[0, 1]$ into $N = 80$ uniform subdivisions $[t_i, t_{i+1}], t = 0, \ldots, N-1$ with $t_0 = 0$ and $t_N = 1$. We then reformulate (5.4) to obtain the following recursive equation over the time steps:

$$S_{t_{i+1}} = S_{t_i}\left(\mu \int_{t_i}^{t_{i+1}} u(s)ds - \frac{\sigma^2}{2}t_i + \sigma \Delta W_{t_{i+1}}\right). \tag{5.5}$$

Each integral in (5.5) is approximated by using Simpson's rule with 3 points. 2000 input functions u_t are sampled from the GRF. To compute the integrals in (5.5), we evaluate these input functions u_t at the endpoints and the midpoints of all subdivisions $[t_i, t_{i+1}]$. Altogether, our training data is the set of 2000 pairs (S_I, U_I) where

$$S_I = \{S_{t_i}\}_{i=1}^N, \quad U_I = \left\{\int_{t_i}^{t_{i+1}} u(s)ds\right\}_{i=0}^{N-1}.$$

This is a challenging test case due to the nonlinearity of the solution operator. To obtain a good approximation, we trained the network with 20000 epochs and used a deep Conv1D model with seven hidden layers for the branch network. Moreover, the combined branch-trunk output is passed through a ten-layer refinement network. The results with 4 different inputs are shown in Fig. 5.15. We can observe that the approximations by the SON have similar shape as the reference solutions. Moreover, we can accurately recover the reference noise band.

5.8 Comparison with vanilla DeepONet

In addition to demonstrating the effectiveness of SON when learning noisy operators, we also compare our method's performance against a vanilla DeepONet. The DeepONet will compare its predictions against the noisy operator output $G(u)(y)$ during training, just

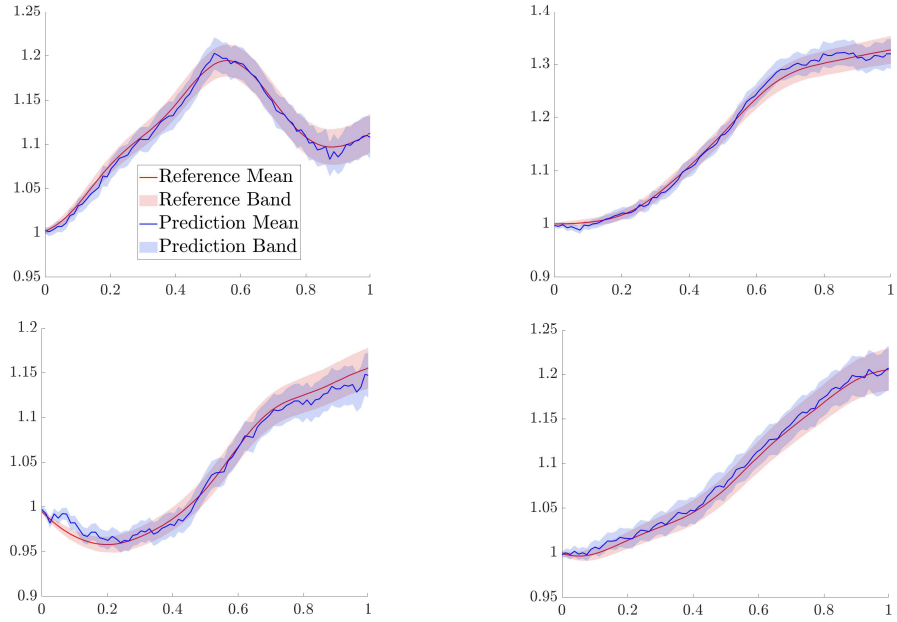

Figure 5.15: Predictions with 4 different inputs.

like SON. Since vanilla DeepONet has no mechanism to probabilistically learn the output noise, all predictions will be fully deterministic.

For the antiderivative operator with noise (5.2), the branch net was a 3 layer sub-network with a consistent 100 hidden neurons per layer while trunk net was a 3 layer sub-network with 64 and 100 hidden neurons per layer. Both had identical neuron output dimensions of 100 each, just like in SON. Vanilla DeepONet achieved final MSEs of 0.016 training and 0.06 testing (Fig. 5.16, left), which is comparable to the ending loss for SON. On its own, this would seem to indicate that the vanilla DeepONet is able to effectively learn the noisy antiderivative operator. However, if we examine the operator output functions for randomly selected u as before (Fig. 5.17), we see that the regular DeepONet only captures the mean of the noisy operator, and fails to capture the noise level in any way. Performing the scaling factor recover test as before confirms this, with an output noise level of $10^{-7} \approx 0$ for floating point. Additionally, we can see that the vanilla DeepONet fails to capture the finer features of the operator output function that have been obscured by noise. Since the noise scaling factor was independent of the operator output domain y, vanilla DeepONet predictions closely align with the actual integral values without noise for many y values. However, it is not clear that this would be the case if the noise scaled differently across the y domain.

Additionally, vanilla DeepONet's performance was also evaluated for the double integral operator with noise (5.5). In this case, the branch net utilized convolutional layers just like applied to the drift and diffusion layer stacks of SON. Vanilla DeepONet achieved final MSEs of 0.002 training and 0.005 testing (Fig. 5.16, right), slightly smaller than for SON. However, since SON introduces randomized noise in the forward propagation, MSE loss

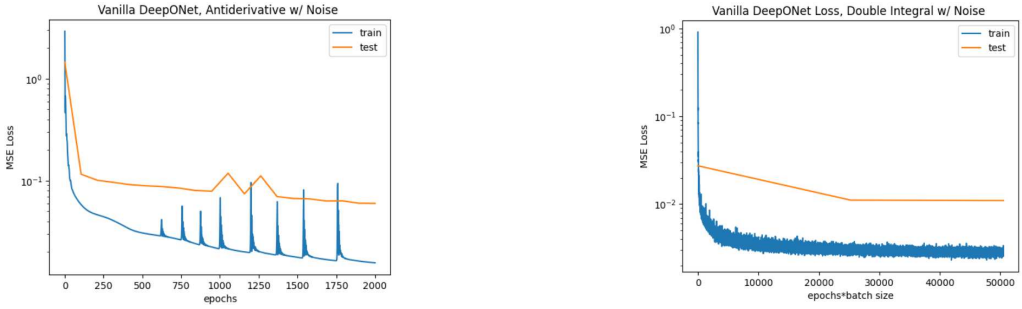

Figure 5.16: Final antiderivative operator MSE losses: 0.016 training, 0.06 testing; Double integral: 0.002 training, 0.005 testing.

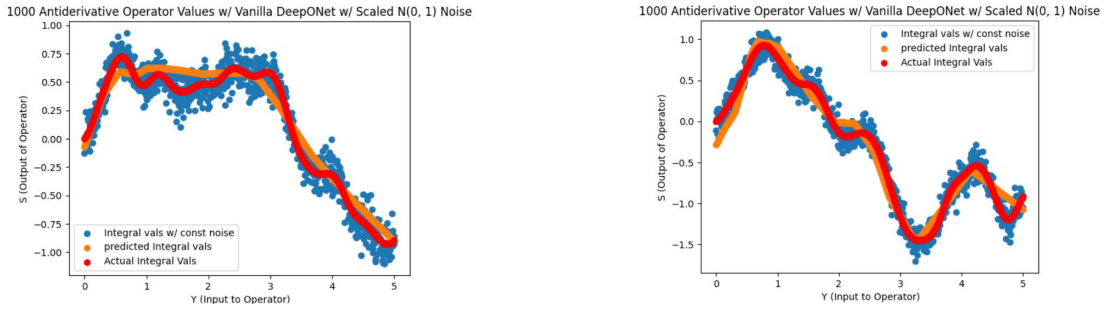

Figure 5.17: Randomly selected antiderivative operator output functions with vanilla DeepONet predictions.

is restricted by the magnitude of that noise band, depending on what values are randomly sampled by the Brownian motion during any given prediction.

Additionally, as expected, vanilla DeepONet fails to quantify the noise level of the operator in its predictions (Fig. 5.18). The DeepONet predictions also struggle to capture the true underlying double integral values, as randomized noise outliers skew the mean of the operator output surface. In Fig. 5.18, this results in the predicted surfaces being slightly above the true double integral values. While random noise outliers also impacted the SON predictions, the quantification of the noise level in the SNN drift layers allowed SON to produce a correctly scaled noise band that includes the true double integral output across a majority of the surface (Fig. 5.11).

We also compare the time to train a vanilla DeepONet versus SON across all experiments. Experiments were run on a Google Colab T4 GPU. Training was for 2000 epochs with a full dataset batch size for the antiderivative, ODE, and 2D ODE operators, while training was 100 epochs with a batch size of 900 for the double integral experiments. The networks also had similar numbers of hidden layers and neurons per hidden layer.

Since SON relies on a much more complicated backpropagation to update the weights, in addition to a more intricate forward process, one might expect it to take significantly longer to train. However, this is not the case, as Table 5.3 demonstrates no clear difference between the two methods. Training SON is just as fast as training DeepONet for the same number of epochs. Of course, this does not take into account the performance of each

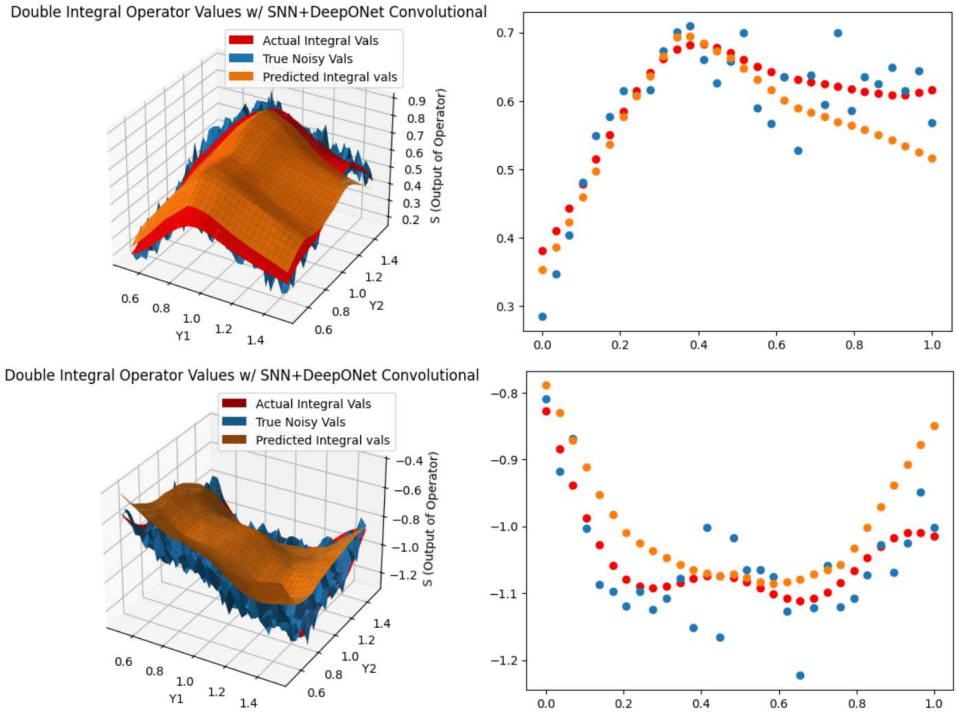

Figure 5.18: Two random vanilla DeepONet double integral operator with noise test set predictions compared with true operator output (left) and corresponding random cross sections (right).

network after a certain number of epochs. SON required 200 epochs to achieve an accurate, convergent performance when predicting the noisy double integral, while the vanilla DeepONet only required 100 to achieve the aforementioned results. Recall however that DeepONet does not quantify noise in the operator while SON does, resulting in a more difficult optimization.

Table 5.3: Training time comparison.

	Antiderivative	ODE	2D ODE	Double integral
DeepONet	6:58	6:47	5:29	13:25
SON	7:01	7:08	7:26	8:05

6 Conclusion

In this work, we presented a novel training mechanism to learn the stochastic functional operators, namely the stochastic operator network. We built on the DeepONet architecture by replacing its branch network with stochastic neural networks, which enables explicit quantification of operator uncertainty. The network's training was then formulated as a stochastic optimal control problem, which was solved by utilizing the stochastic maxi-

mum principle. To demonstrate the effectiveness of our approach, we performed numerical experiments on both integral operators and solution operators arising from stochastic differential equations. The results demonstrated that the SON can accurately replicate the outputs of noisy operators in two- and three-dimensional settings while effectively quantifying operator uncertainty via trainable parameters. Moreover, SON consistently recovered the noise-scaling factor across all experiments, illustrating the strength of our approach for uncertainty quantification during training. We also showed that our new method achieved comparable performance to that of the vanilla DeepONet. More specifically, the SON incurs no significant training penalty while accurately replicating operator noise. Future work includes exploring more complex uncertainty settings for functional operators, applying the SON to challenging SDE and SPDE models, and incorporating data assimilation techniques to extend its applicability to scenarios with sparse training observations.

References

[1] D. Andersson, *Contributions to the Stochastic Maximum Principle*, PhD Thesis, KTH Royal Institute of Technology, 2009.

[2] R. Archibald, F. Bao, Y. Cao, and H. Sun, Numerical analysis for convergence of a sample-wise backpropagation method for training stochastic neural networks, *SIAM J. Numer. Anal.*, 62(2):593–621, 2024.

[3] R. Archibald, F. Bao, Y. Cao, and H. Zhang, A backward SDE method for uncertainty quantification in deep learning, *Discrete Contin. Dyn. Syst. Ser. S* , 15(10):2807–2835, 2022.

[4] R. Archibald, F. Bao, and J. Yong, A stochastic gradient descent approach for stochastic optimal control, *East Asian J. Appl. Math.*, 10(4):635–658, 2020.

[5] K. Bhattacharya, B. Hosseini, N. B. Kovachki, and A. M. Stuart, Model reduction and neural networks for parametric PDEs, *SMAI J. Comput. Math.*, 7:121–157, 2021.

[6] T. Björk, *Arbitrage Theory in Continuous Time*, Oxford University Press, 2019.

[7] L. Bottou, F. E. Curtis, and J. Nocedal, Optimization methods for large-scale machine learning, *SIAM Rev.*, 60(2):223–311, 2018.

[8] R. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, Neural ordinary differential equations, in: *Advances in Neural Information Processing Systems*, Vol. 31, Curran Associates, Inc., 6571–6583, 2018.

[9] T. Chen and H. Chen, Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems, *IEEE Trans. Neural Netw.*, 6(4):911–917, 1995.

[10] E. Dupont, A. Doucet, and Y. W. Teh, Augmented neural ODEs, in: *Advances in Neural Information Processing Systems*, Vol. 32, Curran Associates, Inc., 3140–3150, 2019.

[11] N. Geneva and N. Zabaras, Quantifying model form uncertainty in reynolds-averaged turbulence models with Bayesian deep neural networks, *J. Comput. Phys.*, 383:125–147, 2019.

[12] N. Geneva and N. Zabaras, Modeling the dynamics of PDE systems with physics-constrained deep auto-regressive networks, *J. Comput. Phys.*, 403:109056, 2020.

[13] R. Gerstberger and P. Rentrop, Feedforward neural nets as discretization schemes for ODEs and DAEs, *J. Comput. Appl. Math.*, 82:117–128, 1997.

[14] L. Guo, H. Wu, Y. Wang, W. Zhou, and T. Zhou, IB-UQ: Information bottleneck based uncertainty quantification for neural function regression and neural operator learning, *J. Comput. Phys.*, 510:113089, 2024.

[15] E. Haber and L. Ruthotto, Stable architectures for deep neural networks, *Inverse Problems*, 34:014004, 2018.

[16] J. Jia and A. Benson, Neural jump stochastic differential euquations, in: *Advances in Neural Information Processing Systems*, Vol. 32, Curran Associates, Inc., 9815–9826, 2019.

[17] L. Kong, J. Sun, and C. Zhang, SDE-Net: Equipping deep neural networks with uncertainty estimates, in: *Proceedings of the 37th International Conference on Machine Learning*, PMLR, 119:5405–5415, 2020.
[18] N. B. Kovachki, S. Lanthaler, and A. M. Stuart, Operator learning: Algorithms and analysis, *arXiv: 2402.15715*, 2024.
[19] Y. Kwon, J.-H. Won, B. J. Kim, and M. C. Pail, Uncertainty quantification using Bayesian neural networks in classification: Application to biomedical image segmentation, *Comput. Statist. Data Anal.*, 142:106816, 2020.
[20] V.-A. Le and M. Dik, A mathematical analysis of neural operator behaviors, *arXiv:2410.21481*, 2024.
[21] S. Lee and Y. Shin, On the training and generalization of deep operator networks, *SIAM J. Sci. Comput.*, 46(4):C273–C296, 2024.
[22] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, Fourier neural operator for parametric partial differential equations, *arXiv:2010.08895*, 2020.
[23] X. Liu, T. Xiao, S. Si, Q. Cao, S. Kumar, and C.-J. Hsieh, Neural SDE: Stabilizing neural ODE networks with stochastic noise, *arXiv:1906.02355*, 2019.
[24] X. Liu, T. Xiao, S. Si, Q. Cao, S. Kumar, and C.-J. Hsieh, How does noise help robustness? Explanation and exploration under the neural SDE framework, in: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 279–287, 2020.
[25] L. Lu, P. Jin, and G. E. Karniadakis, Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators, *arXiv:1910.03193*, 2019.
[26] L. Lu, X. Meng, S. Cai, Z. Mao, S. Goswami, Z. Zhang, and G. E. Karniadakis, A comprehensive and fair comparison of two neural operators (with practical extensions) based on fair data, *Comput. Methods Appl. Mech. Engrg.*, 393:114778, 2022.
[27] L. Lu, G. Pang, P. Jin, Z. Zhang, and G. E. Karniadakis, Learning nonlinear operators via deeponet based on the universal approximation theorem of operators, *Nat. Mach. Intell.*, 3:218–229, 2021.
[28] J. Ma and J. Yong, *Forward-Backward Stochastic Differential Equations and Their Applications*, in: *Lecture Notes in Mathematics*, Vol. 1702, Springer, 1999.
[29] P. L. McDermott and C. K. Wikle, Bayesian recurrent neural network models for forecasting and quantifying uncertainty in spatial-temporal data, *Entropy*, 21(2):184, 2019.
[30] M. A. Rahman, M. A. Florez, A. Anandkumar, Z. E. Ross, and K. Azizzadenesheli, Generative adversarial neural operators, *arXiv:2205.03017*, 2022.
[31] A. V. Savchenko, Probabilistic neural network with complex exponential activation functions in image recognition, *IEEE Trans. Neural Netw. Learn. Syst.*, 31:651–660, 2020.
[32] H. Sun, *Stochastic Optimal Control through Gradient Projection Method and Backward Action Learning*, PhD Thesis, Florida State University, 2023.
[33] B. Tzen and M. Raginsky, Neural stochastic differential equations: Deep latent Gaussian models in the diffusion limit, *arXiv:1905.09883*, 2019.
[34] E. Weinan, A propsal on machine learning via dynamical systems, *Commun. Math. Stat.*, 5:1–11, 2017.
[35] L. Wu, K. Zulueta, Z. Major, A. Arriaga, and L. Noels, Bayesian inference of non-linear multiscale model parameters accelerated by a deep neural network, *Comput. Methods Appl. Mech. Engrg.*, 360:112693, 2020.
[36] L. Yang, X. Meng, and G. E. Karniadakis, B-PINNs: Bayesian physics-informed neural networks for forward and inverse PDE problems with noisy data, *J. Comput. Phys.*, 425:109913, 2021.
[37] J. Yao, W. Pan, S. Ghosh, and F. Doshi-Velez, Quality of uncertainty quantification for Bayesian neural network inference, in: *Proceedings at the International Conference on Machine Learning: Workshop on Uncertainty & Robustness in Deep Learning (ICML)*, 2019. `https://finale.seas.harvard.edu/publications/quality-uncertainty-quantification-bayesian-neural-network-inference`
[38] Z. Zhang, W. T. Leung, and H. Schaeffer, Belnet: Basis enhanced learning, a mesh-free neural operator, *arXiv:2212.07336*, 2022.