NEURAL NETWORK APPROXIMATIONS TO SOLUTION OPERATORS FOR PARTIAL DIFFERENTIAL EQUATIONS

by

Nick Winovich

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Mathematics

West Lafayette, Indiana

August 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL STATEMENT OF COMMITTEE APPROVAL

Dr. Guang Lin, Chair

Department of Mathematics, Statistics and School of Mechanical Engineering

Dr. Jianlin Xia

Department of Mathematics

Dr. Karthik Ramani

School of Mechanical Engineering

Dr. Xianfan Xu

School of Mechanical Engineering and Birck Nanotechnology Center

Approved by:

Dr. Plamen Stefanov

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere appreciation and gratitude for the valuable guidance and steadfast support of my supervisor, Dr. Guang Lin. He has served as a constant source inspiration throughout my graduate studies and has provided me with tremendous opportunities and encouragement which have made this work possible.

I would also like to thank Dr. Jianlin Xia, Dr. Karthik Ramani, and Dr. Xianfan Xu for kindly agreeing to serve as members of my defense committee and for the knowledge they have imparted through instruction and research supervision. In addition, I gratefully acknowledge the kind support of Dr. Karthik Ramani and Dr. Carol Handwerker who provided me with the incredibly rewarding opportunity to take part in the IGERT program on the development of sustainable electronics. Finally, I would like to thank Dr. Mohamed Ebeida for his dedicated mentorship, guidance, and support throughout my time spent at Sandia.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In this work, we introduce a framework for constructing light-weight neural network approximations to the solution operators for partial differential equations (PDEs). Using a data-driven offline training procedure, the resulting operator network models are able to effectively reduce the computational demands of traditional numerical methods into a single forward-pass of a neural network. Importantly, the network models can be calibrated to specific distributions of input data in order to reflect properties of real-world data encountered in practice. The networks thus provide specialized solvers tailored to specific use-cases, and while being more restrictive in scope when compared to more generally-applicable numerical methods (e.g. procedures valid for entire function spaces), the operator networks are capable of producing approximations significantly faster as a result of their specialization.

In addition, the network architectures are designed to place pointwise posterior distributions over the observed solutions; this setup facilitates simultaneous training and uncertainty quantification for the network solutions, allowing the models to provide pointwise uncertainties along with their predictions. An analysis of the predictive uncertainties is presented with experimental evidence establishing the validity of the uncertainty quantification schema for a collection of linear and nonlinear PDE systems. The reliability of the uncertainty estimates is also validated in the context of both in-distribution and out-of-distribution test data.

The proposed neural network training procedure is assessed using a novel convolutional encoder-decoder model, ConvPDE-UQ, in addition to an existing fully-connected approach, DeepONet. The convolutional framework is shown to provide accurate approximations to PDE solutions on varying domains, but is restricted by assumptions of uniform observation data and homogeneous boundary conditions. The fully-connected DeepONet framework provides a method for handling unstructured observation data and is also shown to provide accurate approximations for PDE systems with inhomogeneous boundary conditions; however, the resulting networks are constrained to a fixed domain due to the unstructured nature of the observation data which they accommodate. These two approaches thus provide complementary frameworks for constructing PDE-based operator networks which facilitate the real-time approximation of solutions to PDE systems for a broad range of target applications.

1. INTRODUCTION AND BACKGROUND MATERIAL

> Caminante, no hay camino
> sino estellas en la mar.
>
> Antonio Machado

This work aims to summarize the research insights gathered from an investigation into the potential strategies for applying neural network models to the application of partial differential equations (PDEs), which serve as the governing constraints for a broad range of physical systems. More specifically, we examine the possibility of approximating solution operators associated with PDE systems by constructing light-weight neural network surrogate models equipped with an automated form of predictive uncertainty estimation.

This chapter covers the essential material from the three primary fields of study that serve as the foundation of the research presented in this work: artificial neural networks, probability theory, and partial differential equations. An overview of the key concepts and constructions for neural networks is provided in Section 1.1. This is followed by a brief review of probability theory in Section 1.2, with an emphasis placed on the framework of Gaussian process regression. A very short summary of the general theory of partial differential equations, along with a quick review of the finite element method, is provided in Section 1.3. Finally, a review of the related works, historical literature, and the motivation for operator networks is presented in Section 1.4.

1.1 Neural networks

1.1.1 Overview

Neural networks are a class of simple, yet effective, computing systems with a diverse range of applications. These systems comprise large numbers of small, efficient computational units which are organized to form large, interconnected networks capable of carrying out complex calculations. In this section we review the key principles and defining components of neural network architectures which will be used to construct the operator networks proposed later in this work.

1.1.2 Network layers and activation functions

The fundamental building block of feedforward neural networks is the fully-connected neuron illustrated in Figure 1.1[1]. A single fully-connected neuron/unit consists of a collection of *weight parameters* $\{w_\mathrm{i}\}$, equal in number to the input variables $\{x_\mathrm{i}\}$ entering the neuron, along with a *bias parameter* b and an *activation function* f. Once the input variables are specified, the neuron output y is defined by the formula:

$$y \;=\; f\Big(\sum\nolimits_\mathrm{i} w_\mathrm{i}\cdot x_\mathrm{i} \,+\, b\Big) \qquad \text{or} \qquad y \;=\; f(\,\mathbf{w}^T\mathbf{x} \,+\, b\,) \tag{1.1}$$

where $\mathbf{w}$ and $\mathbf{x}$ denote the column vector representations of the neuron weights and inputs.

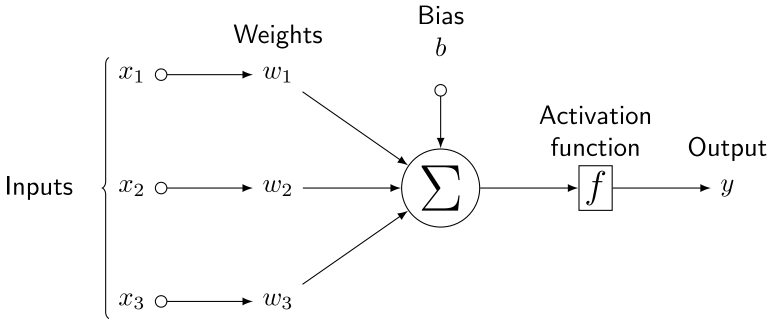


Figure 1.1. Overview of the components of a single artificial neuron.

More generally, a fully-connected *layer* can be formed by processing the input data by multiple independent neurons. These layers produce a vector of output values (often interpreted as *features* extracted from the input data) and are determined by a weight matrix $\mathbf{W} = \{w_\mathrm{ij}\}$ and bias vector $\mathbf{b} = \{b_\mathrm{j}\}$:

$$y_\mathrm{j} \;=\; f\Big(\sum\nolimits_\mathrm{i} w_\mathrm{ij}\cdot x_\mathrm{i} \,+\, b_\mathrm{j}\Big) \qquad \text{or} \qquad \mathbf{y} \;=\; f(\,\mathbf{W}\mathbf{x} \,+\, \mathbf{b}\,) \tag{1.2}$$

[1] ↑This diagram is based off of Gonzalo Medina's response to the following Stack Exchange post: https://tex.stackexchange.com/questions/132444/diagram-of-an-artificial-neural-network

From this description, we see that the network layer corresponds to a vector-to-vector mapping specified by an affine linear operator followed by the elementwise application of a single nonlinear function.

In order to model more complex mappings, *hidden layers* are typically introduced as intermediate steps between the initial network input values and the final network output.

$$\mathbf{x} \xrightarrow{\text{Layer 1}} \mathbf{h}^{(1)} \xrightarrow{\text{Layer 2}} \mathbf{h}^{(2)} \longrightarrow \quad \ldots \quad \longrightarrow \mathbf{h}^{(k-1)} \xrightarrow{\text{Layer k}} \mathbf{y} \tag{1.3}$$

The number of hidden layers is referred to as the network *depth* and the concept is typically visualized as shown in Figure 1.2. Mathematically, these hidden layers simply correspond to a sequence of function compositions where the output of each layer is fed forward and taken as the input to the next layer until the final output is produced. For example, the evaluation of a fully-connected network with three layers is defined by the equations:

$$\mathbf{h}^{(1)} = f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \;\;,\;\; \mathbf{h}^{(2)} = f(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \;\;,\;\; \mathbf{y} = f(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}) \tag{1.4}$$

$$\Longrightarrow \quad \mathbf{y} = f\Big(\mathbf{W}^{(3)}\,\Big[\,f\Big(\mathbf{W}^{(2)}\,\big[f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})\big] + \mathbf{b}^{(2)}\Big)\Big] + \mathbf{b}^{(3)}\Big) \tag{1.5}$$

For comparison with alternative network architectures, we note the connection between a layer with N nodes and a layer with M nodes entails:

FLOPs	Weights
$2\,M\,N$	$M\,N$

(1.6)

The floating point operation (FLOP) [32] count consists of the M addition operations for including the bias terms along with the $M \cdot N$ multiplication operations and $M \cdot (N-1)$ addition operations required for matrix-vector multiplication of the weight matrix with the input/previous-layer values.

The activation functions play an essential role in neural network architectures since they provide the only source of nonlinearity in the resulting models. In particular, composing multiple layers without applying activation functions still results in an affine linear transformation and can therefore be represented by a single layer. Accordingly, the concept of

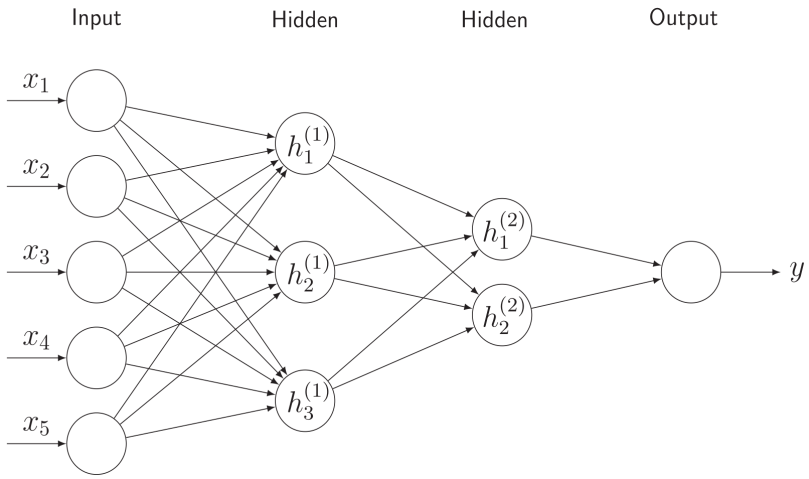


Figure 1.2. Example of neural network defined using intermediate/hidden layers.

network depth is meaningful only in the presence of activation functions, and the proper choice of activation often plays a significant role in the overall performance of the networks.

Two of the most common bounded activation functions are the Sigmoid and hyperbolic tangent (Tanh) functions:

$$\mathrm{Sigmoid}(x) \;=\; \frac{1}{1\,+\,\exp(-x)} \qquad\qquad \mathrm{Tanh}(x) \;=\; \frac{\mathrm{e}^{x}-\mathrm{e}^{-x}}{\mathrm{e}^{x}+\mathrm{e}^{-x}} \tag{1.7}$$

The Rectified Linear Unit (ReLU) and Softplus activations are also commonly used:

$$\mathrm{ReLU}(x) \;=\; \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \qquad\qquad \mathrm{Softplus}(x) \;=\; \log\Big(1\,+\,\exp(x)\Big) \tag{1.8}$$

The Leaky ReLU and and Exponential Linear Unit (ELU) provide parameterized variations of the ReLU activation and can help avoid problems such as “vanishing gradients”[2]:

$$\mathrm{Leaky\_ReLU}_{\alpha}(x) \;=\; \begin{cases} x & x \geq 0 \\ \alpha \cdot x & x < 0 \end{cases} \qquad\qquad \mathrm{ELU}_{\alpha}(x) \;=\; \begin{cases} x & x \geq 0 \\ \alpha \cdot (\mathrm{e}^{x}\,-\,1) & x < 0 \end{cases} \tag{1.9}$$

[2] ↑The problem of “vanishing gradients” refers to situations where the performance of optimization procedures is undermined by neurons producing negative inputs to ReLUs that result in zero-valued gradients.

Conventionally, the values of α in the ELUs and Leaky ReLUs are treated as hyper-parameters (i.e. values specified manually prior to training); it also possible to define activation functions with parameters treated as trainable variables (i.e. values which the network is permitted to change during the learning process), as is the case for Parameterized ReLU [24] and Swish [70] activation functions.

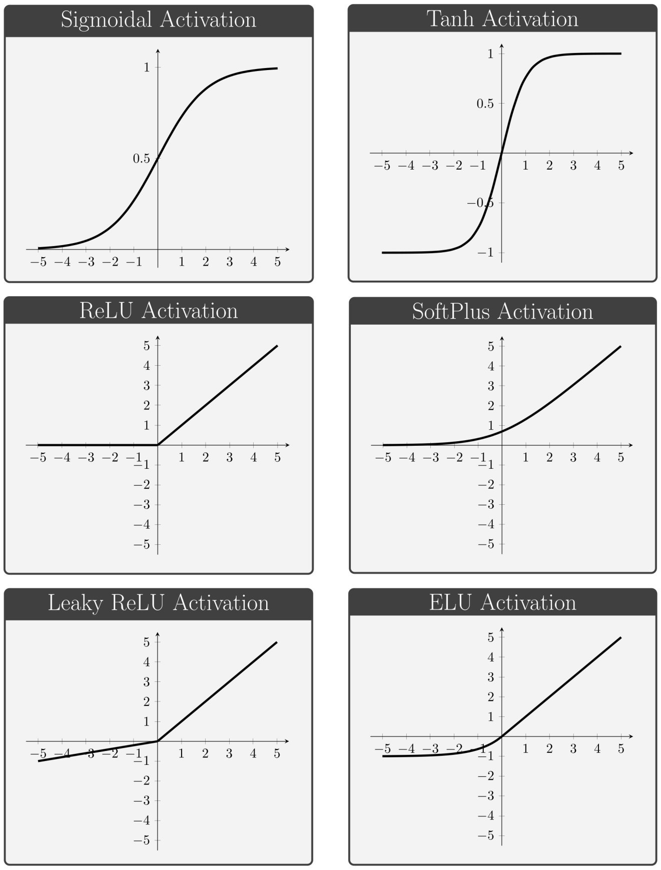


Figure 1.3. Graphs of common activation functions used in neural networks.

The selection of activation functions for hidden layers, as well as the choice of layer sizes and network depth, are typically determined through a series of initial experiments used to identify the settings which yield the best performance. However, the choice of the activation function used for the final network layer, connecting the last hidden layer to the network's final prediction values, requires particular attention. In regression models, for example, the final activations are usually omitted to avoid placing undesirable restrictions on network predictions (e.g. non-negative values produced by ReLU activations).

1.1.3 Universal approximation theorems

In 1989, a number of works were published establishing the approximation capabilities of artificial neural networks, which serve as the basis for a broad collection of results which are commonly referred to as the *universal approximation theorems*. These results extend a number of early works [21, 33, 50, 52] which aimed to explain the observed approximation properties of neural networks under more restrictive assumptions.

Cybenko [14] produced a key result establishing the approximation capabilities of superpositions of sigmoidal functions. Translated into the context of neural networks, this result shows that scalar-valued feedforward networks with one hidden layer and sigmoidal activation functions are universal approximators for continuous functions. More precisely, Cybenko proves that the collection of functions which can be expressed in the following form:

$$\sum_{\mathrm{i}=1}^{N} \omega_{\mathrm{i}} \cdot \sigma(w_{\mathrm{i}}^T x_{\mathrm{i}} \, + \, b_{\mathrm{i}}) \tag{1.10}$$

is dense is the space of continuous functions defined on the unit hypercube in $\mathbb{R}^d$. As noted above, this collection of functions coincides precisely with the definition of feedforward neural networks with one hidden sigmoidal layer followed by a bias-free linear layer.

That same year, Funahashi [19] provided a proof that neural networks with multiple hidden layers are capable of approximating and continuous function defined on a compact subset of $\mathbb{R}^d$ provided that the activation functions used are continuous, non-constant, bounded, and monotonically increasing.

Hornik et al. [30] provided a more general result which proved that multi-layer neural networks are capable of approximating any Borel measurable function[3] defined on a compact subset of $\mathbb{R}^d$ provided that the activation functions $f(x)$ satisfied the following properties: $f : \mathbb{R} \to [0,1]$ is non-decreasing with $\lim_{x\to-\infty} f(x) = 0$ and $\lim_{x\to\infty} f(x) = 1$. This general result provides the key theoretical foundation for neural network models, which have since been employed as universal approximators for a broad range of practical applications.

[3] ↑A brief overview of the concept of Borel measurability is provided in Section 1.2.2

1.1.4 Loss functions and stochastic gradient descent

While the results from the previous section show that neural networks are capable of providing accurate approximations in theory, it still remains to be shown how such networks can be constructed. In particular, the values of the weights and biases must be calibrated to the specific problem under consideration in order for the neural network to produce any meaningful approximation. To this end, it is necessary to provide the network with a form of guidance which precisely defines the problem and quantifies the network's performance. This information is summarized by the network's *loss function* $L(\theta)$ which quantifies the network performance based on the current network parameters θ.

In the context of neural networks, gradient descent appears to provide a reasonable approach for tuning network parameters. In particular, gradient descent gives a simple, iterative algorithm for finding local minima of a real-valued function numerically, which may consider applying to the network loss function $L(\theta)$. The iteration step of the algorithm is defined in terms of a step size parameter α (or by a decreasing sequence $\{\alpha_t\}$) by setting: $\theta_{t+1} = \theta_t - \alpha \cdot \nabla L(\theta_t)$. The initial weights and biases can be interpreted as a single vector θ_0, and the iteration steps from the previous slide could, in theory, be used to identify the optimal parameters θ^* for the model. The issue with this approach is that the function we are actually trying to minimize is defined in terms of the entire dataset $\mathcal{D}$:

$$L(\theta) = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} l(x|\theta) \tag{1.11}$$

where $l(x|\theta)$ denotes the loss for a single example x when using the model parameters θ. In particular, the standard gradient descent algorithm would require computing the average loss (and evaluating the loss for every dataset example) at each step of the iterative scheme.

Since computing the true gradient $\nabla L(\theta)$ at every step is impractical for large datasets, we can instead approximate the gradient using smaller, more manageable batches of data:

$$\nabla L(\theta) \approx \nabla \widehat{L}_t(\theta) = \frac{1}{|\mathcal{B}_t|} \sum_{x \in \mathcal{B}_t} \nabla l(x|\theta) \tag{1.12}$$

where the batches $\{\mathcal{B}_t\}$ partition the dataset into smaller subsets (typically of equal size). The iteration step for the stochastic batch gradient descent algorithm is then defined by replacing the full gradient $\nabla L(\theta)$ with the approximate gradient $\nabla \widehat{L}_t(\theta)$ at each step. This procedure is often referred to as mini-batch gradient descent, a particular form of stochastic gradient descent (SGD) where the stochasticity in the optimization procedure is a result of the dataset subsampling used to approximate gradients.

Since fixed learning rates often lead to suboptimal performance in practice, it is typically necessary to gradually reduce the learning rate gradually during the optimization procedure. However, specifying the proper learning rate "schedule" is complicated by the facts that:

- Manually tuning learning rates for each application is time consuming and imprecise.
- In many cases, different parameters require entirely different learning rates.

One method for calibrating learning rates based on information gathered from previous steps is to incorporate a notion of momentum into the update policy:

$$\theta_{t+1} = \theta_t - v_t \ \text{ where } \ v_t = \gamma \cdot v_{t-1} + \alpha \cdot \nabla L(\theta) \tag{1.13}$$

An accelerated form of incorporating momentum was introduced by Nesterov [64] in 1983 which leverages the value of "looking ahead" before making updates:

$$\theta_{t+1} = \theta_t - v_t \ \text{ where } \ v_t = \gamma \cdot v_{t-1} + \alpha \cdot \nabla L(\theta - \gamma \cdot v_{t-1}) \tag{1.14}$$

More recently, the AdaGrad [16] and RMSProp [27] algorithms have been proposed as improved forms of momentum-based SGD. Both of these methods make use of a common technique for estimating momentum incrementally at each iterative step. In particular, the methods make use of exponential moving averages which are obtained by applying an exponential decay to the values from previous iterations so that an emphasis is placed on the most recent values; this allows the average to move, or correct itself, as the distribution of the values changes.

For example, to track the gradient $g_t = \nabla L(\theta_{t-1})$ of the loss with respect to the parameters θ, we can define an average recursively by setting:

$$\begin{cases} m_0 \; = \; 0 \\ m_t \; = \; \beta \cdot m_{t-1} + (1-\beta) \cdot g_t \end{cases} \quad \Longrightarrow \quad m_t \; = \; (1-\beta) \sum_{\tau=1}^{t} \beta^{t-\tau} \cdot g_\tau \tag{1.15}$$

where the parameter $\beta < 1$ is used to specify the exponential decay rate and is typically taken to be close to 1. While this technique typically works well, the zero-initialization $m_0 = 0$ is rather arbitrary. In the following section, an algorithm will be introduced which is specifically designed to avoid the undesirable effects resulting from this choice of initialization.

1.1.5 Adam optimizer

Among the most commonly used variations on conventional SGD is the *Adam* optimization algorithm, which was introduced by Kingma and Ba [36] and derives its name from the concept of "adaptive moment estimation". As the name suggests, the intuition behind the algorithm is that the efficiency of a gradient descent based optimization scheme can be improved by tracking the moments[4] of the gradient estimates and incorporating this information into the update policy of the numerical scheme. In particular, the Adam algorithm tracks the first two uncentered moments of the objective function gradient at each step:

$$\{m_t \, , \, v_t\} \quad \text{with} \quad \mathbb{E}[m_t] \, \approx \, \mathbb{E}[\nabla_\theta L_t(\theta)] \quad \text{and} \quad \mathbb{E}[v_t] \, \approx \, \mathbb{E}[(\nabla_\theta L_t(\theta))^2] \tag{1.16}$$

where $L_t(\theta)$ denotes the evaluation of the fixed loss function on the particular batch of data occurring at time-step t. This information is incorporated into the gradient descent algorithm by rescaling the step-size of each parameter update by a factor of $m_t/\sqrt{v_t}$ at each step. This is motivated by the intuition that $m_t/\sqrt{v_t}$ can, loosely speaking, be interpreted as a signal-to-noise ratio for the stochastic gradient estimates. Accordingly, the step-size of the descent scheme is automatically reduced in regions where the gradient estimates are becoming increasingly noisy. This is of particular importance when the scheme has arrived

[4] ↑The measure-theoretic definitions of expectation and moments are provided in Section 1.2.2

near a true minimum of the objective function; at such a minimum, the gradient estimates will simply reduce to mean-zero noise (caused by evaluation at varying subsamples of the full dataset). In the context of the Adam optimization algorithm, the optimal parameters θ^* are taken to be those which minimize the expected value of the objective function; i.e. $\theta^* = \operatorname{argmin}_\theta \mathbb{E}[L(\theta)]$ where the expectation is taken with respect to the batch data $\{\mathcal{D}_t\}$.

An outline of the implementation details of the Adam algorithm is provided in Figure 1.4. The main idea is to track the moment estimates by maintaining two exponential moving averages $\{m_t\}$ and $\{v_t\}$. Of note, however, is the fact that the moment estimates are biased by the fact that they have been arbitrarily initialized to zero. Fortunately, this bias can be easily accounted for using the observation that:

$$m_t \;=\; (1-\beta_1) \sum_{\tau=1}^{t} \beta_1^{t-\tau} \cdot g_\tau \;\text{ and }\; v_t \;=\; (1-\beta_2) \sum_{\tau=1}^{t} \beta_2^{t-\tau} \cdot g_\tau^2 \tag{1.17}$$

where g_τ denotes the gradient estimate $\nabla_\theta L_\tau(\theta_{\tau-1})$ at time-step τ; these equalities follow from the exponential moving average construction and a simple expansion of the recursive definitions of m_t and v_t. In the case where the true first and second moments are stationary, i.e. $\mathbb{E}[g_\tau]$ and $\mathbb{E}[g_\tau^2]$ are constant at each step, it follows that:

$$\mathbb{E}[m_t] \;=\; \mathbb{E}\left[(1-\beta_1) \sum_{\tau=1}^{t} \beta_1^{t-\tau} \cdot g_\tau\right] \;=\; \mathbb{E}[g_t](1-\beta_1) \sum_{\tau=1}^{t} \beta_1^{t-\tau} \;=\; \mathbb{E}[g_t] \cdot (1-\beta_1^t) \tag{1.18}$$

$$\mathbb{E}[v_t] \;=\; \mathbb{E}\left[(1-\beta_2) \sum_{\tau=1}^{t} \beta_2^{t-\tau} \cdot g_\tau^2\right] \;=\; \mathbb{E}[g_t^2](1-\beta_2) \sum_{\tau=1}^{t} \beta_2^{t-\tau} \;=\; \mathbb{E}[g_t^2] \cdot (1-\beta_2^t) \tag{1.19}$$

From this, we observe that the bias introduced by the zero-initialization can be removed from the moment estimates by defining the adjusted estimates:

$$\widehat{m}_t \;=\; \frac{m_t}{1-\beta_1^t} \qquad \widehat{v}_t \;=\; \frac{v_t}{1-\beta_2^t} \tag{1.20}$$

By eliminating the initialization bias used for the exponential moving average calculations, the Adam algorithm can lead to considerable improvements in performance; this optimization algorithm will be used for all of the neural network training procedures presented in this work.

```
Algorithm 1.   Adam Optimization Algorithm
Require: Objective function $L(\theta)$, initial parameters $\theta_0$, stepsize hyperparameter $\alpha$,
    exponential decay rates $\beta_1, \beta_2$ for moment estimates, tolerance parameter $\lambda > 0$
    for numerical stability, and decision rule for declaring convergence of $\theta_t$ in scheme.
procedure Adam($L,\ \theta_0\ ;\ \alpha,\ \beta_1,\ \beta_2$)
    $m_0,\ v_0,\ t \leftarrow [0,\ 0,\ 0]$                          # Initialize moment estimates
                                                        # and timestep to zero
    # Begin optimization procedure
    while $\theta_t$ has not converged do
        $t \leftarrow t + 1$                                     # Update timestep
        $g_t \leftarrow \nabla_\theta L_t(\theta_{t-1})$                          # Compute gradient of objective
        $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$            # Update first moment estimate
        $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (g_t \odot g_t)$   # Update second moment estimate
        $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$                     # Create unbiased estimate $\widehat{m}_t$
        $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$                      # Create unbiased estimate $\widehat{v}_t$
        $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \lambda)$      # Update objective parameters
    return $\theta_t$                                           # Return final parameters
```

Figure 1.4. Psuedocode for the Adam optimization algorithm; based off of Algorithm 1 from the paper where Kinga and Ba introduced the algorithm [36].

1.1.6 Backpropagation

At this point, we have defined the computational structure of neural networks, established their capacity for function approximation after proper parameter tuning with respect to a given loss function, and specified procedures for tuning the network parameters accordingly. However, one fundamental question regarding the optimization of neural networks still remains: *"How are the gradients of network parameters actually computed?"*.

The concept of backpropagation [75], first introduced in 1986, provides a natural solution to this question in the context of neural networks. The authors considered the context of a neural network with sigmoidal activations σ and a loss E defined in terms of the network predictions y_{j} and true values d_{j} by the following equation:

$$E \;=\; \frac{1}{2} \sum_{\mathrm{j}} (y_{\mathrm{j}} - d_{\mathrm{j}})^2 \tag{1.21}$$

As shown in Figure 1.5, the gradient information computed at the final stage of the neural network architecture can be "propagated backward" to determine gradients associated with network quantities earlier on in the network. The influence of the value x_j, for example, can be gauged by computing the gradients $\frac{\partial E}{\partial x_j}$ by means of the Chain Rule:

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_j} = \frac{\partial E}{\partial y_j} \cdot \frac{d}{dx_j}[\sigma(x_j)] \tag{1.22}$$

Likewise, the gradients with respect to the weights w_{ji} and previous layer outputs y_i can be computed via:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial x_j} \cdot \frac{\partial x_j}{\partial w_{ji}} = \frac{\partial E}{\partial x_j} \cdot y_i \tag{1.23}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} \cdot \frac{\partial x_j}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} \cdot w_{ji} \tag{1.24}$$

Now that the error contribution associated with y_i is known, the contributions from network parameters of the previous layer can be computed using the same methodology that was applied to y_j. In this way, gradient calculations for all network parameters can be computed by propagating back the error contributions through each of the network layers.

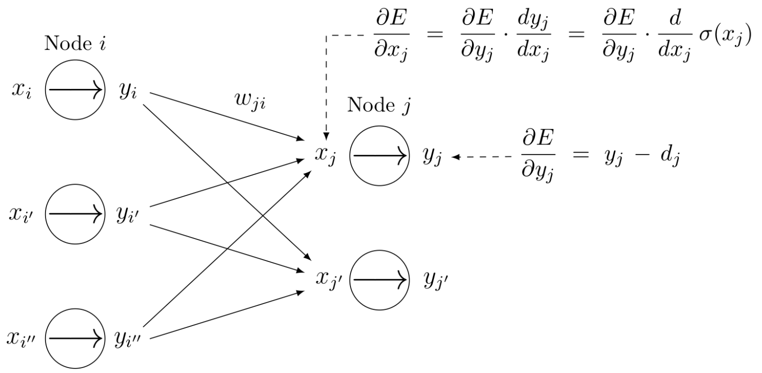

$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \cdot \frac{dy_j}{dx_j} = \frac{\partial E}{\partial y_j} \cdot \frac{d}{dx_j}\sigma(x_j)$
Node i
x_i
y_i
w_{ji}
Node j
x_j
y_j
$\frac{\partial E}{\partial y_j} = y_j - d_j$
$x_{i'}$
$y_{i'}$
$x_{j'}$
$y_{j'}$
$x_{i''}$
$y_{i''}$

Figure 1.5. Overview the backpropagation technique for computing the gradients required by gradient descent in the context of neural networks.

1.1.7 Automatic differentiation

While the backpropagation framework provides us with an overall strategy for computing the gradients of the network error with respect to each of the network parameters, the derivative calculations themselves require further attention. Two commonly used methods for automating the process of computing derivatives are *symbolic differentiation* and *numeric differentiation*; however, both of these methods have severe practical limitations in the context of training neural networks:

- Symbolic differentiation produces exact derivatives through direct manipulation of the mathematical expressions used to define functions; the resulting expressions can be lengthy and contain unnecessary computations, however, and are inefficient unless additional “expression simplification” steps are included.

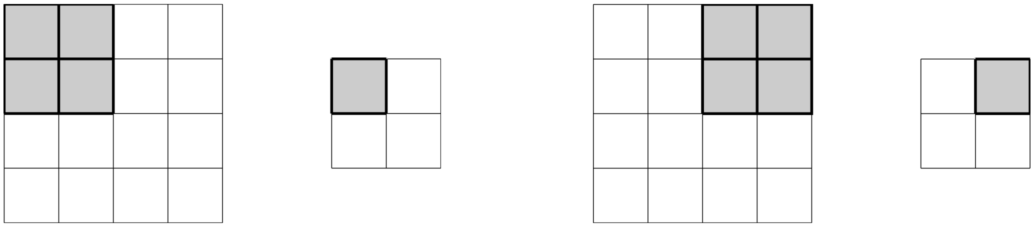
- Numeric differentiation techniques are widely applicable and efficient; however, the resulting inexact gradient estimates can entirely undermine the training process.

Automatic differentiation (AD) [9] in “reverse mode” provides a generalization to backpropagation and gives us a way to carry out the required gradient calculations *exactly* and *efficiently*. This approach computes derivatives using the underlying computational graph associated with the specified network structure. In particular, a trace of all elementary operations required for computing the network output is stored on an evaluation “tape”, or “Wengert list”, and common operations are shared to avoid repeating calculations that are common to both the evaluation and derivative calculations [11].

1.1.8 Convolutional architectures

In many cases the spatial orientation of the input data plays an important role in the correct interpretation of the data. Fully-connected layers are extremely inefficient at learning spatial connections, however, since the ordering/arrangement of the data has no influence in the overall network architecture (in the sense that if all entries of the input dataset were permuted in a fixed manner, the performance of the network would remain unchanged).

Convolutional layers are specifically suited for processing data with spatial features (e.g. images, time-series, etc.). The fundamental idea behind convolutional layers is to restrict the domain of influence of any given input; in particular, convolutional layers specify a receptive field which determines which output neurons are affected by each input value. As illustrated in Figure 1.6, convolutional layers operate by applying a *filter* which is designed to slide across the input array to produce output values; this is done by multiplying filter weights with the input values covered by the current filter position. A *stride* is also typically introduced, corresponding to the number of steps the filter slides after each calculation; in the example below the filter starts in the upper-left to produce the output value y_{11}, shifts two steps right to align with the upper-right entries used to produce the value y_{12}, then shifts two steps down to repeat the process on the lower entries of the array.



$$y_{11} \;=\; f(w_{11}x_{11} + w_{12}x_{12} + w_{21}x_{21} + w_{22}x_{22}) \qquad\qquad y_{12} \;=\; f(w_{11}x_{13} + w_{12}x_{14} + w_{21}x_{23} + w_{22}x_{24})$$

Figure 1.6. Overview of the computations prescribed by a convolutional network layer.

As mentioned in the discussion of fully connected layers, matrix representations offer a natural way of expressing the connection between network layers. Convolutional layers are also easily expressed in matrix form, however the dense weight matrices from fully-connected layers are replaced with highly structured, sparse matrices. For example, the matrix representation for applying a convolutional layer with a 2 × 2 filter and stride 2 to a 4 × 4 input layer is given by:

$$\begin{bmatrix} y_{11} \\ y_{12} \\ y_{21} \\ y_{22} \end{bmatrix} = f\left(\begin{bmatrix} w_{11} & w_{12} & 0 & 0 & w_{21} & w_{22} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & w_{11} & w_{12} & 0 & 0 & w_{21} & w_{22} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & w_{11} & w_{12} & 0 & 0 & w_{21} & w_{22} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & w_{11} & w_{12} & 0 & 0 & w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} | \\ \widehat{\mathbf{x}} \\ | \end{bmatrix} \right) \tag{1.25}$$

$$\text{where} \quad \widehat{\mathbf{x}} \;=\; \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} & x_{21} & x_{22} & x_{23} & x_{24} & x_{31} & x_{32} & x_{33} & x_{34} & x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix}^T \tag{1.26}$$

The concept of convolution also generalizes naturally to "multi-channel" arrays (for example, image data defined by RGB channels associated with different colors). A convolutional layer between an input array with N channels and an output array with M channels is defined by a collection of $N \cdot M$ distinct filters, with associated weight matrices $\mathbf{W}^{(n,m)}$ for

$n \in \{1, \ldots, N\}$ and $m \in \{1, \ldots, M\}$, corresponding to the connections between each input channel and each output channel. In addition, each output channel is also assigned a bias term, $b^{(m)} \in \mathbb{R}$ for $m \in \{1, \ldots, M\}$, and the final outputs for the m^{th} channel are given by:

$$\mathbf{y}^{(m)} \;=\; f\left(\textstyle\sum_{n=1}^{N} \mathbf{W}^{(n,m)}\mathbf{x}^{(n)} \,+\, b^{(m)}\right) \tag{1.27}$$

In some applications it is necessary to restore the original resolution of the input array. This requires an upsampling technique designed to increase the resolution of an array. It is often possible to upsample directly using a technique such as bilinear interpolation. The concept of convolutional layers also admits a natural generalization, however, which provides an alternative method for upsampling and is referred to as a transpose convolutional layer. As the name suggests, the transpose convolutional layer is defined by a weight matrix corresponding to the transpose of a standard convolutional weight matrix.

From the above matrix representation, we note that a stride 1, $k \times k$ convolutional layer between a layer with resolution $R \times R$ and N channels and a layer with M channels entails:

FLOPs	Weights
$\approx\; 2\,k^2\,R^2\,N\,M$	$k^2\,N\,M$

(1.28)

Here the FLOP calculation is carried out as follows:

$$k^2\,R^2\,N\,M \;+\; (k^2-1)\,R^2\,N\,M \;+\; R^2\,(N-1)\,M \;+\; R^2\,M \;=\; 2\,k^2\,R^2\,N\,M \tag{1.29}$$

The first term corresponds to the multiplication operations with the filter weights in each filter position, the second term corresponds to the addition operations for each position, the third term accounts for the addition operations between the filtered input channels, and the last term accounts for the addition operations for including bias terms. This calculation is only approximate since it does not account for padding considerations (which will typically reduce the overall count by a negligible amount). It is typically important to keep the filter size k relatively small in order to maintain a manageable number of trainable variables and FLOPs. However, we note that the number of variables and FLOPs associated with convolutional layers are substantially smaller than those required by fully-connected layers (in particular, since the input data consists of R^2 values, the connections required by a fully-connected layer between two such arrays would be on the order of R^4).

In addition to providing an efficient form of processing spatially structure data, convolutional architectures also provide powerful tools for generating and transforming structured data. One of the most prolific design strategies used for modeling data transformations is to construct an architecture according to an *encoder-decoder* structure. As illustrated in Figure 1.7, these architectures consist of an "encoder" component designed to map the original data into a lower-dimensional "latent-space" followed by a "decoder" component which maps the extracted latent features back to a structured output/prediction. These architectures have been well-established in the computer vision community for applying transformations to image and video data with a broad range of applications; we will also adopt this structure in Chapter 2 where encoder-decoder models are used to model function-to-function mappings.

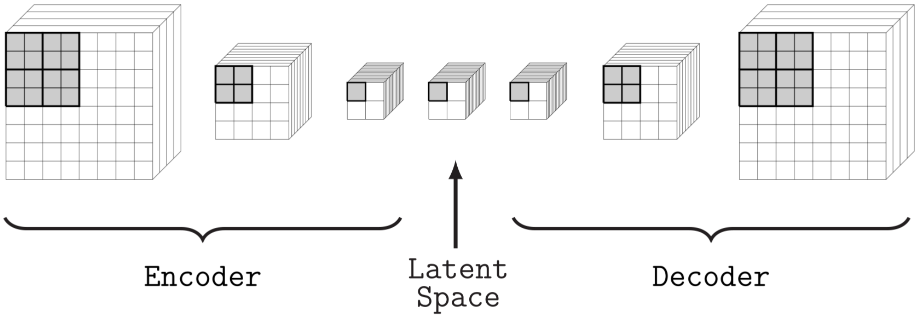


Figure 1.7. Overview of a typical encoder-decoder structure used for convolutional architectures.

1.2 Probability theory

1.2.1 Overview

Probability theory provides a powerful theoretical framework for handling uncertainties involved with mathematical models and systems. For the purposes of this work, we will restrict our attention to a few specific concepts which serve as the theoretical motivation for the network architectures proposed in subsequent chapters. The material covered in this section serves as the basis for the proposed works in two fundamental ways:

1. Gaussian processes serve as the primary motivation/inspiration for the concept of predictive uncertainty and the definition of the neural network loss functions.

2. The datasets used for the proposed neural network training procedure are constructed using samples from Gaussian processes.

For a more detailed analysis of the topics covered in this section, the following works are recommended: *Gaussian Processes for Machine Learning* [71], *Machine Learning: A Probabilistic Perspective* [63], and *Kernel-based Approximation Methods using MATLAB* [18].

1.2.2 Measure theory background

The precise definitions of the concepts involved with probability theory are formulated in the context of measure theory. For completeness, the formal definitions of several key constructions in probability theory are outlined below. A more detailed treatment of this material can be found in *Probability & Measure Theory* [8], and heuristic descriptions of these concepts are also provided in Section 1.2.3 for a more informal review.

A σ-algebra on a set $\mathcal{W}$ is defined to a be a collection of sets $\mathscr{F} \subseteq \mathscr{P}(\mathcal{W})$ (where $\mathscr{P}(\mathcal{W})$ denotes the set of all subsets of $\mathcal{W}$) subject to the constraints:

$$\mathcal{W} \in \mathscr{F} \quad , \quad A \in \mathscr{F} \iff A^C \in \mathscr{F} \quad , \quad \{A_n\}_{n\in\mathbf{N}} \subset \mathscr{F} \implies \bigcup_{n\in\mathbf{N}} A_n \in \mathscr{F} \tag{1.30}$$

In the context of probability theory, such σ-algebras are interpreted as the possible "events" that may occur in a given system or model. The probability of each event (i.e. the likelihood that the event occurs) is determined by specifying a *probability measure* on the space $(\mathcal{W}, \mathscr{F})$: namely, a set function $\mu : \mathscr{F} \to [0, \infty]$ which satisfies the following three properties:

$$\mu(\emptyset) \;=\; 0 \;\;,\;\; \mu(\mathcal{W}) \;=\; 1 \;\;,\;\; \mu\Big(\bigcup_{n\in\mathbf{N}} A_n\Big) \;=\; \sum_{n\in\mathbf{N}} \mu(A_n) \;\text{ for all }\; \{A_n\} \subset \mathscr{F} \text{ disjoint} \tag{1.31}$$

Since these underlying spaces are often rather abstract in practice, the majority of results in probability theory are formulated in terms of functions between spaces which have more intuitive interpretations. In order to develop a consistent analytical framework, the functions considered must behave well with respect to the source and target σ-algebras, and this notion is formalized by the concept of measurability. In particular, given two measurable spaces $(\mathcal{W}_1, \mathscr{F}_1)$ and $(\mathcal{W}_2, \mathscr{F}_2)$, a mapping $f : \mathcal{W}_1 \to \mathcal{W}_2$ is said to be $(\mathscr{F}_1, \mathscr{F}_2)$-*measurable* if $f^{-1}(A) := \{\omega \in \mathcal{W}_1 \,|\, f(\omega) \in A\} \in \mathscr{F}_1$ for every $A \in \mathscr{F}_2$.

A *random variable* on a measurable space $(\mathcal{W}, \mathscr{F})$ is a real-valued mapping $X : \mathcal{W} \to \mathbb{R}$ which is measurable with respect to the Borel σ-algebra $\mathscr{B}(\mathbb{R})$ on $\mathbb{R}$, which is defined to be the smallest σ-algebra containing all open sets on the real line. Given a random variable X on a probability space $(\mathcal{W}, \mathscr{F}, \mathbb{P})$, we define the *cumulative distribution function* (c.d.f.) of X to be the mapping $F_X : \mathbb{R} \to [0, 1]$ defined for all $t \in \mathbb{R}$ by:

$$F_X(t) \;:=\; \mathbb{P}(X \leq t) \;=\; \mathbb{P}\Big(\{\omega \in \mathcal{W} : X(\omega) \leq t\}\Big) \tag{1.32}$$

In cases where $F_X(t)$ is absolutely continuous, the derivative $p_X(t) := \frac{d}{dt} F_X(t)$ exists almost everywhere on the real line and is referred to as the *probability density function* (p.d.f.) of the random variable X. This allows us to recast many of the key constructs from probability spaces in terms integration of standard functions on the real line:

$$\mathbb{P}[X \in E] \;=\; \int_{x\in E} p_X(x)\; dx \qquad \text{for all} \quad E \in \mathscr{B}(\mathbb{R}) \tag{1.33}$$

The *expectation* $\mathbb{E}[X]$ of a random variable X gives a precise formulation of the "average" or "mean". While the precise definition of expectation requires a formal measure-theoretic construction, in scenarios where $p_X(x)$ exists the expectation of X can be computed by:

$$\mathbb{E}[X] \; = \; \int_{\mathbb{R}} x \cdot p_X(x) \; dx \tag{1.34}$$

The higher-order *moments* $\mathbb{E}[X^n]$ can be computed in a similar fashion; notably, the second-order "centered" moment $\text{Var}\,[X] \; := \; \mathbb{E}[(X - \mathbb{E}[X])^2]$ is commonly referred to as the *variance* of the random variable X and can be interpreted as a measure of how much the variable deviates from its mean value.

The relationships between two random variables X and Y are generally specified with respect to the construction of an abstract probability space which is beyond the scope of this work. However, under suitable circumstances as joint density $p_{X,Y}(x, y)$ can be specified which characterizes the likelihood of the events $\{X = x\}$ and $\{Y = y\}$ both occurring (in particular, accounting for potential dependencies between the two random variables). This leads to the notion of the *conditional distribution* $X \,|\, Y$ which formalizes the concept of making conclusions concerning the variable X based off of information which is known about the variable Y. This notion of conditioning plays a critical role in the context of inference, since it allows us to incorporate observation data into probabilistic models and refine the model predictions as more data is made available.

1.2.3 Random variables and probability distributions

The two core components of probability theory that will be relevant for this work are concepts of *random variables* and *stochastic processes*. Intuitively, a random variable can simply be interpreted as a value which unknown until a certain event occurs or a measurement is made; the range in possible values, as well as the likelihood of a given value occuring, is characterized by the *distribution* of the random variable. Under certain assumptions, the distribution of a random variable can be characterized by a probability density function; examples of the densities that will be considered in this work are illustrated in Figure 1.8.

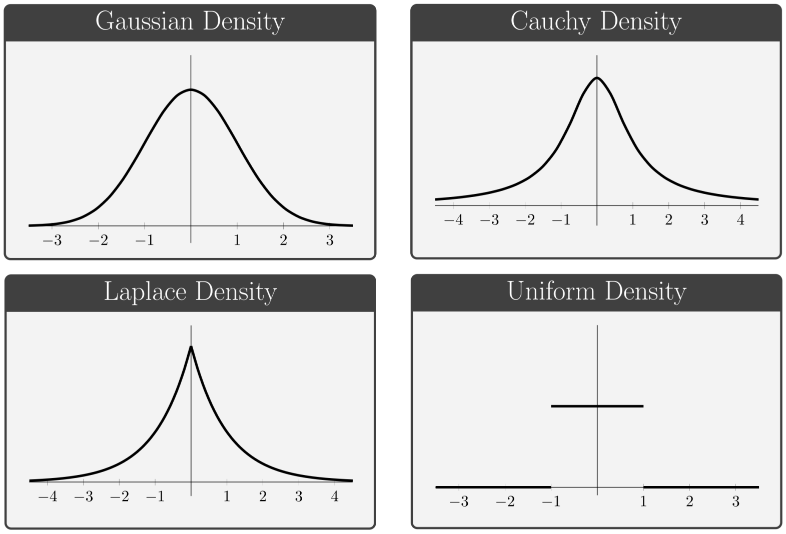


Figure 1.8. Graphs of common probability density functions.

Distribution	Density Function	Parameters
Gaussian	$p_X(x) \;=\; \frac{1}{\sqrt{2\pi\sigma^2}}\,\exp(-\frac{1}{2}(x-\mu)^2/\sigma^2)$	μ, σ
Cauchy	$p_X(x) \;=\; (\pi\gamma\,(1+(x-\widetilde{\mu})^2/\gamma^2))^{-1}$	$\widetilde{\mu}, \gamma$
Laplace	$p_X(x) \;=\; \frac{1}{2b}\,\exp(-\lvert x-\mu)\rvert/b)$	μ, b
Uniform	$p_X(x) \;=\; \frac{1}{b-a}\,\mathbb{1}_{[a,b]}(x)$	a, b

Figure 1.9. Expressions for common probability density functions.

Among the most commonly occurring distributions in practice, and one which will play a central role in the results presented in this work, is the Gaussian, or Normal, distribution. A Gaussian random variable X is defined by a continuous density of the form:

$$p_X(x) \;=\; \frac{1}{\sqrt{2\pi\sigma^2}}\;\exp\Big(-(x-\mu)^2/\sigma^2\Big) \qquad \forall\; x \in \mathbb{R} \tag{1.35}$$

and is uniquely determined by two parameters: the *mean* μ and the *variance* $\sigma^2 > 0$. To specify the distribution more concisely, the notation $X \sim \mathcal{N}(\mu, \sigma^2)$ is commonly used. This density can also be rewritten in the following form, which motivates a natural generalization to higher dimensions:

$$p_X(x) \;=\; (2\pi)^{-1/2}\;|\sigma^2|^{-1/2}\;\exp\Big(-(x-\mu)(\sigma^2)^{-1}(x-\mu)\Big) \tag{1.36}$$

The concept of Gaussian random variables also extends naturally to higher-dimensional quantities referred to as multivariate Gaussian random variables, or Gaussian random vectors. A multivariate Gaussian random variable X has a density of the form:

$$p_X(x) \;=\; (2\pi)^{-d/2}\,|\Sigma|^{-1/2} \;\; \exp\Big(-(x-\mu)^T\Sigma^{-1}(x-\mu)\Big) \quad \forall\; x \in \mathbb{R}^d \tag{1.37}$$

The random variable is defined by two defining parameters: the *mean vector* $\mu \in \mathbb{R}^d$ and the *covariance matrix* $\Sigma \in \mathbb{R}^{d\times d}$. This is only a well-defined density if the covariance matrix Σ is symmetric and *positive-definite*: i.e. $x^T\,\Sigma\,x \;>\; 0 \quad \forall\; x \in \mathbb{R}^d$. Equivalently, all eigenvalues of the matrix Σ must be positive. Notably, a random vector $X \;=\; (X_1, \,\ldots\,, X_d)^T$ is Gaussian if and only if every linear combination of its components is Gaussian:

$$\text{i.e.} \qquad \sum_{\mathrm{i}=1}^{d} \alpha_{\mathrm{i}}\, X_{\mathrm{i}} \;\sim\; \mathcal{N}(\mu_\alpha, \sigma_\alpha^2) \qquad \forall\; \alpha \in \mathbb{R}^d \tag{1.38}$$

In particular, each component X_{i} is Gaussian, though this is not sufficient to ensure that the random vector X is Gaussian. By construction, the matrix Σ is the covariance matrix between the components of the vector X: i.e. $\mathrm{Cov}[X_{\mathrm{i}}, X_{\mathrm{j}}] \;=\; \Sigma_{\mathrm{ij}}$[5].

In the context of inference, situations often arise where the values of some vector components are known while others are not. This gives rise to a natural partition of the component values consisting of the unknown values $X^{(1)} \;=\; \{X_1, \,\ldots\,, X_r\}$ and the observed values $X^{(2)} \;=\; \{X_{r+1}, \,\ldots\,, X_d\}$. In this case the covariance matrix takes the form:

$$\Sigma \;=\; \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \quad \text{where} \quad \Sigma_{\mathrm{ij}} = \mathrm{Cov}[X^{(\mathrm{i})}, X^{(\mathrm{j})}] \tag{1.39}$$

[5] ↑The *covariance* between random variables provides a general measure of the correlation or relationship between the variables and is defined by $\mathrm{Cov}\,[X_{\mathrm{i}}, X_{\mathrm{j}}] \;:=\; \mathbb{E}[(X_{\mathrm{i}} - \mathbb{E}[X_{\mathrm{i}}])(X_{\mathrm{j}} - \mathbb{E}[X_{\mathrm{j}}])$

Importantly, the resulting conditional distribution $X^{(1)} \,|\, X^{(2)} \;\sim\; \mathcal{N}(\mu^*, \Sigma^*)$ is also normally distributed; moreover, the updated parameters can be computed explicitly using Schur complements [23] and are given by the following formulas:

$$\begin{cases} \mu^* \;=\; \mu^{(1)} - \Sigma_{12}\,\Sigma_{22}^{-1}[X^{(2)} - \mu^{(2)}] \\ \Sigma^* \;=\; \Sigma_{11} \,-\, \Sigma_{12}\,\Sigma_{22}^{-1}\,\Sigma_{12}^T \end{cases} \tag{1.40}$$

In this way, we are able to update/refine our knowledge regarding the unknown values $X^{(1)}$ based off of the observed values $X^{(2)}$.

1.2.4 Gaussian processes and uncertainty estimates

While the formal definition of a stochastic process is beyond the present scope, the following informal interpretation will suffice for our purposes: a stochastic process generalizes the notion of a random vector $X = [X_1, \dots, X_d]$ by replacing the finite index set $\{1, \dots, d\}$ with a domain Ω; namely, for each point $x \in \Omega$, a stochastic process specifies a random variable X_x, just as a random vector specifies a random variable X_{i} for each $\mathrm{i} \in \{1, \dots, d\}$.

A continuous stochastic process $X : \Omega \to \mathbb{R}$ on a domain $\Omega \subset \mathbb{R}^d$ is said to be a Gaussian process if for every finite collection of points $\{x^{(1)}, \dots, x^{(k)}\}$ the random vector $(X(x^{(1)}), \dots, X(x^{(k)}))$ is a multivariate Gaussian. The notation $X_x = X(x)$ is also often used, and will be adopted here to emphasize the dependence on outcomes from an underlying probability space $\mathcal{W}$; in particular, $X_x(\omega)$ is used to denote the observed value of the process X at spatial location $x \in \Omega$ for a given outcome $\omega \in \mathcal{W}$.

Gaussian processes are completely determined by a *mean function* $\mu(x)$ and *covariance function* $K(x, y)$. The mean function must be continuous, and the covariance function must be symmetric and positive definite in the sense that the matrix $[K(x_{\mathrm{i}}, x_{\mathrm{j}})]_{\mathrm{i,j}=1}^N$ is positive semi-definite for all $N \in \mathbb{N}$ and $\{x^{(1)}, \dots, x^{(N)}\}$ in Ω.

From a probabalistic standpoint, a Gaussian process specifies a distribution over a space of functions defined on the domain Ω [71]. In particular, for each outcome ω in the underlying probability space, the process X defines a continuous function $X_*(\omega) : \Omega \to \mathbb{R}$ by mapping $x \mapsto X_x(\omega)$ for all $x \in \Omega$.

While the mean function $\mu(x)$ influences the general trends of the sample paths/functions associated with a given Gaussian process, the covariance function $K(x, y)$ dictates the high-level characteristics and regularity properties. In many situations, it is reasonable to assume that the correlation between function values at any two evaluation locations x and y is determined solely by the separation distance $r = \|x - y\|$. This often motivates the use of *stationary isotropic* covariance functions $K(x, y) = \kappa(\|x-y\|)$, which are defined with respect to a scalar-valued function $\kappa : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ and encompass a broad range of commonly used covariance functions [18]. A brief list of common kernels along with the associated mean function properties is provided in Figure 1.10.

Kernel	Expression	Mean Function Property
Matern C^0	$\kappa(r) \;=\; \exp(-r)$	Continuous
Matern C^2	$\kappa(r) \;=\; (1+r)\exp(-r)$	Twice Differentiable
Squared Exp.	$\kappa(r) \;=\; \exp(-r^2/2)$	Smooth

Figure 1.10. Examples of common covariance kernels for Gaussian processes and the corresponding properties of the sample means. Each of these kernels is stationary isotropic, with the covariance between two points x and y determined entirely by the distance $r \;=\; \|x - y\|$.

Once the mean and covariance functions of a Gaussian process are specified, samples paths can be drawn and evaluated at arbitrary spatial locations $\{x_{\mathrm{i}}\}_{\mathrm{i}=1}^{N}$ by assembling the covariance matrix $\Sigma \;=\; [K(x_{\mathrm{i}}, x_{\mathrm{j}})]_{\mathrm{i,j}=1}^{N}$, computing the associated Cholesky factorization, and multiplying the Cholesky factor by samples drawn from a standard multivariate normal distribution [71].

If we are given a set of observations of the process at the points $X_{obs} \;=\; \{x_{\mathrm{i}}^*\}_{\mathrm{i}=1}^{k}$, we can sample values from the updated process at a different set of points $X_{eval} \;=\; \{x_{\mathrm{i}}\}_{\mathrm{i}=1}^{n}$ using the fact that $X_{eval}\,|\,X_{obs} \;\sim\; \mathcal{N}(\mu^*, \Sigma^*)$ with updated parameters given by:

$$\begin{cases} \mu^* \;=\; \mu_{eval} - \Sigma_{12}\,\Sigma_{22}^{-1}[X_{obs} - \mu_{obs}] \\[2ex] \Sigma^* \;=\; \Sigma_{11} \;-\; \Sigma_{12}\,\Sigma_{22}^{-1}\,\Sigma_{12}^{T} \end{cases} \tag{1.41}$$

where $\Sigma_{11} = \mathrm{Cov}[X_{eval}, X_{eval}]$, $\Sigma_{12} = \mathrm{Cov}[X_{eval}, X_{obs}]$, and $\Sigma_{22} = \mathrm{Cov}[X_{obs}, X_{obs}]$ are calculated using the covariance kernel.

In practice, however, the precise form and parameter values for the underlying covariance kernel are typically unknown. In some cases the general form of the covariance function can be inferred from the observations by inspection, but in almost all cases a more precise analysis is required in order to adequately tune the kernel parameters θ. Since the likelihood $p(X_{obs} \,|\, \theta) \sim \mathcal{N}(0, \Sigma)$ of the observation values is given explicitly by:

$$p(X_{obs} \,|\, \theta) \;=\; (2\pi)^{-d/2} \, |\Sigma|^{-1/2} \;\; \exp\left(- X_{obs}^T \Sigma^{-1} X_{obs} \right) \tag{1.42}$$

the associated negative log marginal likelihood (NLML) is given by:

$$\mathrm{NLML} \;=\; -\log p(X_{obs} \,|\, \theta) \;=\; \tfrac{d}{2} \, \log(2\pi) \;+\; \tfrac{1}{2} \, \log(|\Sigma|) \;+\; \tfrac{1}{2} \, X_{obs}^T \Sigma^{-1} X_{obs} \tag{1.43}$$

By minimizing this expression, the optimal values for the kernel parameters θ can be determined so that the Gaussian process model is correctly calibrated to the observation data in considerations; in particular, the term $\frac{1}{2}\log(|\Sigma|)$ accounts for the model's complexity while the term $\frac{1}{2} X_{obs}^T \Sigma^{-1} X_{obs}$ accounts for the model's fit to the data [63]. Once the model parameters have been calibrated, the model predictions and associated uncertainty estimates can be computed using Equation 1.41; importantly, this allows us fit Gaussian process models to potentially noisy data while providing a quantitative measure of model uncertainty along with the mean predictions, as illustrated in Figure 1.11.

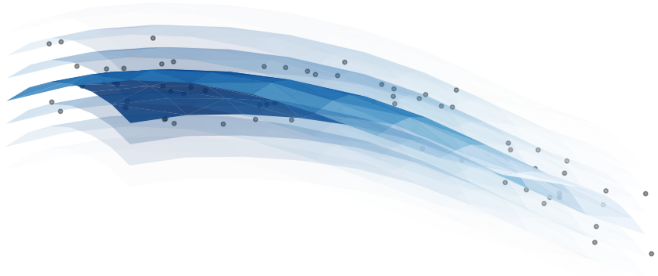


Figure 1.11. Predictive uncertainty estimates (shown as transparent layers) provided by a Gaussian process approximation to noisy observation data.

In the conventional implementation of Gaussian process learning algorithms, scalability is limited due to the following calculations: $\Sigma^{-1}X_{obs}$ and $\log\left(|\Sigma|\right)$. These calculations can be carried out using a Cholesky factorization of the observation covariance matrix Σ, however this requires $\mathcal{O}(n^3)$ operations and $\mathcal{O}(n^2)$ storage when modeling n observations [23]. Once the Cholesky factorization is computed, the cost for inference of a single test point is $\mathcal{O}(n)$ with an additional cost of $\mathcal{O}(n^2)$ for the associated variance estimate. This computational complexity often poses a significant challenge for employing these techniques for real-time or data intensive applications of Gaussian process inference.

However, the automated uncertainty estimates provided by these models often provide valuable insights into the confidence/accuracy of the model predictions. This essential property of Gaussian processes provides us with the fundamental template for equipping neural networks with predictive uncertainties. In addition to motivating the construction of predictive uncertainties, Gaussian processes will be used as a key element of the construction of training data for the proposed neural network models.

1.3 Partial differential equations

1.3.1 Overview

The theory of partial differential equations (PDEs) is central to our understanding of diverse range mathematical systems. Broadly speaking, the study of PDEs involves the analysis of systems where multivariate functions are determined implicitly by relationships imposed on their partial derivatives. These types of systems arise naturally in a variety of contexts and often provide the governing equations for modeling physical, biological, and financial systems.

While the exact solutions of systems of differential equations are often intractable analytically, a wide range of effective numerical techniques exist for computing approximate solutions. For the purposes of this work, we will restrict our attention to one specific numerical method for solving PDEs which is referred to as the finite element method (FEM). In particular, we note that the FEM framework is relevant to the neural networks proposed in this work in two crucial ways: (1) it will be employed to generate the approximate solutions used for the neural network training datasets in Chapter 2 and (2) the concept of FEM basis functions serves as a natural motivation for the network architecture used in Chapter 3.

1.3.2 Second order equations

For concreteness, we will restrict our attention to the context of second order PDEs defined on bounded two-dimensional domains $\Omega \subset \mathbb{R}^2$; this will be the problem setting under consideration for the remainder of this work. The primary task in this context is to identify an unknown function $u : \Omega \to \mathbb{R}$ based on equations which involve second-order derivatives, along with lower order derivatives and the function itself. In general, second order PDEs can be expressed using a single equation of the form:

$$F\left(x, u(x),\, \tfrac{\partial}{\partial x_1}u(x), \tfrac{\partial}{\partial x_2}u(x),\, \tfrac{\partial^2}{\partial x_1^2}u(x),\, \tfrac{\partial}{\partial x_1}\tfrac{\partial}{\partial x_2}u(x),\, \tfrac{\partial^2}{\partial x_2^2}u(x)\right) \;=\; 0 \quad \text{for all} \quad x \in \Omega \tag{1.44}$$

Once the definition of F is specified, our goal is to find a function $u(x)$ which satisfies this equation throughout the domain Ω. Under certain assumptions regarding the structure and regularity of the function F, it can be guaranteed that such a function exists, and any such function is referred to as a *solution* of the PDE. In order to guarantee the uniqueness of a solution, these systems typically include boundary conditions (BCs) which constrain the solutions behavior on the boundary $\partial\Omega$ of the domain [17]. Among the most common forms of boundary conditions are the Dirichlet BCs, characterized by constraints of the form $u(x) = g(x)$, and Neumann BCs, expressed in the form $\frac{\partial}{\partial \vec{\mathbf{n}}} u(x) = g(x)$ (where $\vec{\mathbf{n}}$ denotes the unit outward normal vector along the boundary). The boundary conditions are referred to as homogeneous when $g(x) \equiv 0$ and are referred to as inhomogeneous otherwise.

Many common families of PDEs can be defined by specifying a fixed structure for the function F where variations are parameterized by functions involving only the variable x. For example, the Poisson equation with homogeneous Dirichlet boundary conditions is defined with respect to the Laplace operator $\Delta := \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2}$ by the system:

$$\begin{cases} \Delta u(x) \;=\; f(x) & \text{for all} \quad x \in \Omega \\ \;\; u(x) \;=\; 0 & \text{for all} \quad x \in \partial\Omega \end{cases} \tag{1.45}$$

In this case, the interior constraint can be expressed equivalently by Equation 1.44 with the function F taken to be of the form:

$$F(x, U, U_1, U_2, U_{11}, U_{12}, U_{22}) \;=\; (U_{11} + U_{22}) \;-\; f(x) \tag{1.46}$$

The function $f(x)$ is typically referred to as the source or forcing term of the PDE, and under mild assumptions regarding Ω it can be shown that a unique solution $u(x)$ exists for each $f(x)$ with sufficient regularity. In particular, this assertion holds for any function $f \in C^\infty(\overline{\Omega})$ defined on a bounded, connected domain Ω with a smooth boundary; moreover, it can be shown the the solution also belongs to $C^\infty(\overline{\Omega})$ under these assumptions. From this, it follows that a solution operator $\mathcal{G} : C^\infty(\overline{\Omega}) \to C^\infty(\overline{\Omega})$ exists which is defined by mapping each source term $f(x)$ to the associated solution $u(x)$.

1.3.3 Finite element method

In this section we provide a very brief overview of the finite element method which is primarily intended to serve as a motivation for the neural network models introduced in later chapters. A more detailed analysis can be found in *Partial Differential Equations with Numerical Methods* [51].

The FEM approach is formulated with respect to a collection of *finite elements* consisting of a partition of the underlying domain along with a collection of *basis functions* $\{\varphi_{\mathrm{i}}\}_{\mathrm{i}\in\mathcal{I}}$. The calculations performed by FEM procedures are then confined to the subspace $V_h = \operatorname{span}\{\,\varphi_{\mathrm{i}}\,\}_{\mathrm{i}\in\mathcal{I}}$ induced by the choice of basis functions.

For concreteness, we will restrict our attention to the Poisson equation with homogeneous Dirichlet boundary conditions, as defined in Equation 1.45. By multiplying by an unknown "test" function $\varphi(x)$ on both sides and integrating the resulting equation, an application of integration by parts yields the related, but less strict equation:

$$\int_\Omega \nabla u(x)\cdot\nabla\varphi(x)\;dx \;=\; \int_\Omega f(x)\cdot\varphi(x)\;dx \tag{1.47}$$

The left-hand-side of the equation above motivates the definition of the bilinear form $a(-,-)$ associated with the Laplace operator Δ from the original PDE system:

$$a(\varphi,\psi) \;=\; \int_\Omega \nabla\varphi(x)\cdot\nabla\psi(x)\;dx \quad \forall\,\varphi,\psi\in V_h \tag{1.48}$$

and leads us to what is commonly referred to as the *weak formulation* of the problem:

$$\text{Find}\;\; u_h\in V_h \quad \text{such that} \quad a(u_h,\,v_h) \;=\; \int_\Omega f\cdot v_h\;dx \quad \forall\,v_h\in V_h \tag{1.49}$$

In particular, we aim to find coefficients $\{x_{\mathrm{i}}\}_{\mathrm{i}\in\mathcal{I}}$ for the basis functions φ_{i} so that the associated approximation $u_h = \sum_{\mathrm{i}\in\mathcal{I}} x_{\mathrm{i}}\,\varphi_{\mathrm{i}}$ satisfies:

$$a\left(u_h,\;\varphi_{\mathrm{j}}\right) \;=\; a\left(\textstyle\sum_{\mathrm{i}\in\mathcal{I}} x_{\mathrm{i}}\,\varphi_{\mathrm{i}},\;\varphi_{\mathrm{j}}\right) \;=\; \int_\Omega f\cdot\varphi_{\mathrm{j}}\;dx \quad \forall\,\mathrm{j}\,\in\,\mathcal{I} \tag{1.50}$$

Since the form $a(-,-)$ is bilinear, the central expression can be rewritten as $a\left(\sum_{\mathrm{i}\in\mathcal{I}} x_{\mathrm{i}}\,\varphi_{\mathrm{i}},\, \varphi_{\mathrm{j}}\right) = \sum_{\mathrm{i}\in\mathcal{I}} x_{\mathrm{i}}\cdot a\left(\varphi_{\mathrm{i}},\, \varphi_{\mathrm{j}}\right)$. From this we see that the coefficients $\{x_{\mathrm{i}}\}$ can be interpreted as weighting factors for the values $a(\varphi_{\mathrm{i}},\varphi_{\mathrm{j}})$ for each fixed $\mathrm{j} \in \mathcal{I}$. The full list of constraints defined by Equation 1.50 can therefore be expressed more concisely in matrix form as:

$$Ax \;=\; b \quad \text{where} \quad A_{\mathrm{ij}} \;:=\; a(\varphi_{\mathrm{i}},\varphi_{\mathrm{j}}) \;\; \forall\, \mathrm{i},\mathrm{j} \,\in \mathcal{I} \quad \text{and} \quad b_{\mathrm{j}} \;:=\; \int_{\Omega} f\cdot\varphi_{\mathrm{j}}\; dx \;\;\; \forall\, \mathrm{j}\,\in\,\mathcal{I} \tag{1.51}$$

After constructing the system defined by Equation 1.51 based off of the PDE in consideration, the optimal coefficients $\{\widehat{x}_{\mathrm{i}}\}_{\mathrm{i}\in\mathcal{I}}$ are computed by solving the linear system and an approximate solution $\widehat{u}(x)$ for the PDE is then given by:

$$\widehat{u}(x) \;=\; \sum\nolimits_{\mathrm{i}\in\mathcal{I}} \widehat{x}_{\mathrm{i}}\,\varphi_{\mathrm{i}}(x) \;\;\; \forall\, x\,\in\,\Omega \tag{1.52}$$

Once the basis functions $\{\varphi_{\mathrm{i}}\}_{\mathrm{i}\in\mathcal{I}}$ are specified, the overall objective of the finite element method can thus be framed in the context of mapping input data $f(x)$ to the correct coefficients $\{\widehat{x}_{\mathrm{i}}\}_{\mathrm{i}\in\mathcal{I}}$ in order to produce an accurate approximation $\widehat{u}(x)$ to the solution. This interpretation provides a key source of inspiration for the "operator networks" that will be considered in this work; in particular, we will aim to model the high-level mappings of the form "$f \mapsto \widehat{u}$" directly using neural network architectures.

1.4 Related works and operator networks

1.4.1 Historical background and related works

Following the universal approximation results of the late 1980's, a number of strategies were proposed aiming to leverage the approximation capacity of neural networks in the context of PDEs [15, 47, 53, 78]. Of particular relevance for the present work, is the 1992 paper submitted by Dissanayake and Phan-Thien entitled *Neural-Network-Based Approximations for Solving Partial Differential Equations* [15].

The work of Dissanayake and Phan-Thien showed that neural networks could used to approximate the solution of the following PDE system:

$$\begin{cases} \Delta u(x) &= \sin(\pi x_1)\cdot\sin(\pi x_2) \quad \forall\, x\in[0,1]^{\times 2} \\ u(x) &= 0 \quad\quad\quad\quad\quad\quad\quad\quad\quad \forall\, x\in\partial[0,1]^{\times 2} \end{cases} \tag{1.53}$$

and that the same method could be applied to obtain an approximate solution for a nonlinear problem on the unit square as well.

The neural network architectures used in this case were very simple feedforward networks with small dense layers and sigmoidal activation functions. More specifically, the authors conducted experiments using two hidden layers and 3 to 10 units per layer, as illustrated in Figure 1.12. Of note is the fact that the input to the neural network consists of a single spatial location $x = (x_1, x_2)$ and the output consists of a single scalar value intended to provide an approximation to the solution value $u(x)$ at the prescribed input location.

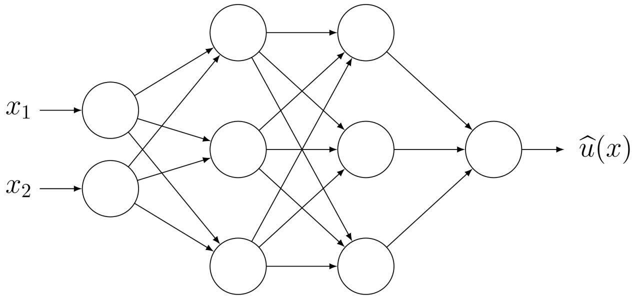


Figure 1.12. A neural network architecture capable of approximating the solution $u(x)$ of a fixed PDE, as proposed by Dissanayake and Thien [15].

In order to capture the model fit with respect to the PDE, the network loss function is defined using point-collocation to approximate the L^2 error of the approximation $\widehat{u}(x)$ on the interior and boundary of the domain:

$$\text{Loss} \;=\; \Big(\textstyle\sum_{x\in\Omega_d} |\Delta\widehat{u}(x) \,-\, \Delta u(x)|^2\Big)^{1/2} \;+\; \Big(\textstyle\sum_{x\in\partial\Omega_d} |\widehat{u}(x) \,-\, u(x)|^2\Big)^{1/2} \tag{1.54}$$

where Ω_d and $\partial\Omega_d$ denote discrete sets of points on a coarse grid used to model the interior and boundary of the domain, respectively. The gradients of the loss function with respect to the network parameters, as well as the Laplacian values $\Delta\widehat{u}(x)$, are then derived manually using backpropagation. A quasi-Newtonian method, such as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, is then used to tune the network weights by minimizing the loss associated with the neural network approximations.

More recently, a number of more complex neural network models have been proposed for obtaining numerical solutions to PDEs [15, 47, 48, 53, 61, 76, 78, 79, 81]. A substantial amount research has also been directed toward the development of a physics-informed deep learning framework for modeling physical data with governing constraints defined by PDEs [68, 69]. As was the case in the work of Dissanayake and Phan-Thien, these neural network frameworks have focused primarily on the construction of specialized networks, with each network dedicated to a specified PDE. This setup is motivated by the universal approximation property [14, 30, 31] of neural networks, with the trained network associated with a fixed PDE intended to serve as an embodiment of the solution itself. These networks are individually tailored to the specific problem and geometry into consideration, and by construction must be retrained for each new system. In effect, the neural network training procedure is employed as a substitute for explicitly forming and solving the linear system corresponding to the FEM framework.

One shortcoming of the conventional FEM approach, and the more recent neural network PDE solvers, however, is that neither provides any form of uncertainty estimate associated with their proposed solutions. This can result in overconfidence in the accuracy of the approximate solutions produced by the solvers, potentially leading to a reliance on inaccurate approximations without any indication of the risk. In an effort to mitigate this vulnerability, works in the field of probabilistic numerics have aimed to design Bayesian frameworks for approximating the solutions to PDEs using Gaussian process priors [12, 26]. These solvers provide both mean and variance estimates, corresponding to a Bayesian posterior distribution associated with the solver's approximate solutions.

This Bayesian framework has the advantage of providing a natural form of uncertainty quantification; in particular, when uncertainties begin to accumulate in the numerical algorithm, and the approximation has the potential for significant inaccuracies in a certain region, the resulting uncertainty in the predicted solution is clearly indicated in the form of high variance estimates.

While the probabilistic numerics approaches provide accurate, rigorous uncertainty estimates, they also inherently rely on the explicit conditioning of Gaussian processes; this leads to cubic computational complexity and results in extremely slow inference speeds. The neural network-based methods, on the other hand, provide exceptionally fast estimates but do so without any attempt to quantify the associated uncertainty. To incorporate the benefits of uncertainty quantification into deep learning models, Gal and Ghahramani [20] introduced a dropout procedure for representing model uncertainty without incurring the computational costs of fully Bayesian models. More recently, Lakshminarayanan et al. [49] introduced an ensemble method for predicting uncertainty estimates using deterministic networks, and Kendall and Gal [35] proposed a detailed framework for modeling epistemic and aleatoric uncertainties in deep learning models using Bayesian neural networks (BNNs) and loss functions derived from likelihood calculations.

Zhu and Zabaras [83] introduced a fully convolutional BNN designed to predict solutions to stochastic PDEs in an image-to-image manner; i.e. input arrays specifying the terms of the PDE are mapped directly to the associated solution array. Importantly, this setup allows the model to make predictions for new systems without requiring the model to be retrained. Moreover, since the network weights are realized as random variables in the Bayesian framework, uncertainty estimates are naturally provided by Monte Carlo approximations to the pointwise variance of the predicted solution (using samples from the posterior distributions of the network weights). The variational inference procedure used to train this network is specifically designed to account for the limited availability of training data on a fixed domain in consideration; generalizing this model to solve PDEs on varied domains requires extremely large data sets and would be infeasible due to the computational demands of the Bayesian training procedure.

1.4.2 Motivation for operator networks

The majority of existing neural network models are designed to take advantage of the universal approximation property of neural networks in a fairly straight-forward manner: the goal is essentially to tune the network weights/parameters so that the resulting, 'trained' network provides a faithful surrogate model for an unknown/intractable solution function.

In the context of function regression/inference, for example, the input to the neural network is typically assumed to be coordinate data 'x' and the output of the network is interpreted as an approximation to the solution value '$u(x)$' at the location specified by the input, as shown in Figure 1.12. After training, the neural network surrogate can be evaluated quickly at any input location to provide an approximation to the true solution function; however, whenever the problem statement is modified (e.g. when a source term is changed) the target solution changes as well and the network must therefore be retrained, as depicted in Figure 1.13.

As noted in Section 1.3.2, it is also possible to view the task of solving PDEs in the context of a solution operator $\mathcal{G} : C^\infty(\overline{\Omega}) \to C^\infty(\overline{\Omega})$ which assigns to each admissible input function $f(x)$ the associated solution $u(x)$. This motivates the possibility of constructing neural networks which are designed to model the high-level function-to-function mappings '$f \mapsto \widehat{u}$' associated with PDE systems instead of modeling the pointwise mappings '$x \mapsto \widehat{u}(x)$' for each individual problem instance. This concept serves as the primary focus of the work presented in the following chapters; namely, the construction of "operator networks' designed to compute approximate solutions to PDE systems. Importantly, the resulting models do not require any retraining between problem instances which provides a fundamental advantage over conventional 'approximator networks' in the context of real-time applications.

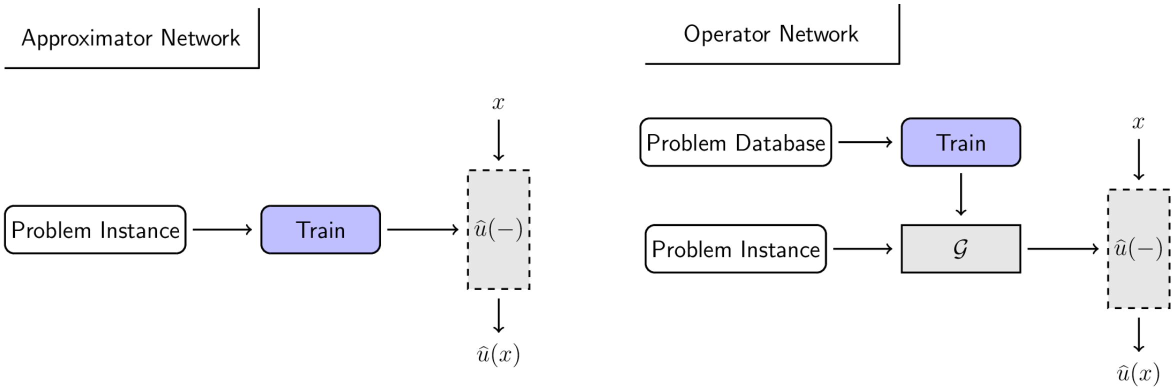


Figure 1.13. Comparison of training and inference workflow for ‘approximator networks’ (left) designed to model pointwise mappings and ‘operator networks’ (right) designed to model high-level mappings of functions.

2. APPROXIMATING OPERATORS WITH CONVOLUTIONS

> La généralisation est l'un des meilleurs moyens de «fair comprendre» en mathématiques.
>
> Henri Lebesgue

2.1 Introduction

In this chapter, we introduce a data-driven neural network framework, referred to as ConvPDE-UQ [82], for constructing numerical PDE solvers which provide accurate uncertainty estimates and are applicable to varied domains/geometries. These solvers are implemented as fully convolutional networks and are trained offline using a diverse collection of FEM solutions to PDEs on varying domains. To take advantage of recent advances in deep learning for computer vision [74, 77], an image-to-image network architecture is introduced to construct a solver which does not require retraining for new problems or different domains; once training is complete, approximate solutions can be obtained using a single forward-pass of the convolutional network. A theoretical justification for the approximation of direct PDE input-to-solution mappings by neural networks is established based on the existence and properties of Green's functions. In addition, a probabilistic training procedure is proposed which casts network predictions as pointwise Gaussian estimates and leads to a natural form of uncertainty quantification with virtually no additional computational demands. The performance of the proposed framework is demonstrated on a collection of three distinct classes of PDEs consisting of two linear elliptic problems and a nonlinear Poisson problem. The inference speed of these solver networks are shown to be 30 times faster than the parallelized FEM implementations using FEniCS [1, 2, 3, 4, 5, 6, 7, 28, 29, 34, 38, 39, 40, 41, 42, 43, 44, 45, 54, 55, 56, 57, 58, 59, 66, 67, 72, 73]. Moreover, a careful analysis of the network uncertainty estimates is carried out, and the interpretation of the network predictions as pointwise Gaussian estimates is shown to be strongly supported by the experimental evidence.

2.2 Problem setup

2.2.1 Mathematical framework

To begin, we consider the task of solving the Poisson equation with the Dirichlet boundary condition:

$$\begin{cases} \Delta u \;=\; f & \text{in} \quad \Omega \\ \;\; u \;=\; 0 & \text{on} \quad \partial\Omega \end{cases} \tag{2.1}$$

where the domain $\Omega = D$, corresponding to the unit disk in $\mathbb{R}^2$, is fixed and we aim to solve the system for a family of *source terms* $f \in C^\infty(\overline{\Omega})$. The solution to this problem can be obtained explicitly in terms of the fundamental solution to the Laplace operator $\Delta = \sum_{\mathrm{i}=1}^{2} \frac{\partial^2}{\partial x_{\mathrm{i}}^2}$ and the Green's function associated with the domain Ω. We recall that the fundamental solution Γ to the Laplacian in $\mathbb{R}^2$ is given by:

$$\Gamma(x) \;=\; \frac{1}{2\pi} \ln(|x|) \quad \forall\, x \,\in\, \mathbb{R}^2 \setminus \{0\} \tag{2.2}$$

and the Green's function for the unit disk in $\mathbb{R}^2$ is defined by:

$$G(x,y) \;=\; \begin{cases} \Gamma(x-y) \;-\; \Gamma\Big(\, |y|\, (x-\overline{y}) \Big) & \text{for} \quad y \neq 0 \\ \Gamma(x) & \text{for} \quad y \,=\, 0 \end{cases} \tag{2.3}$$

where $\overline{y} = y/|y|^2$ corresponds to a scaled reflection of interior points $y \in D \setminus \{0\}$ across the boundary of the disk [22]. The solution to the homogeneous Dirichlet problem posed above can then be expressed using the representation formula:

$$u(x) \;=\; \int_\Omega G(x,y)\, f(y)\; dy \quad \forall\, x \,\in\, \Omega \tag{2.4}$$

Accordingly, we have an explicit solution mapping:

$$\mathcal{G} \ : C^\infty(\overline{\Omega}) \ \longrightarrow \ C^\infty(\overline{\Omega}) \tag{2.5}$$

$$f \ \mapsto \ \textstyle\int_\Omega G(-,y)f(y)dy \tag{2.6}$$

which assigns to each admissible source term f the associated solution $\mathcal{G}f = u$. This solution mapping takes the form of an integral operator with kernel $G(x,y)$ and satisfies the well-known *maximum principle* [22]: the unique solution $u \in C^\infty(\overline{\Omega})$ to Equation (2.1) with source term $f \in C^\infty(\overline{\Omega})$ satisfies:

$$\sup_\Omega |\,\mathcal{G}f\,| \ \leq \ C \cdot \sup_\Omega |f| \tag{2.7}$$

In particular, the solution mapping $\mathcal{G} : C^\infty(\overline{\Omega}) \to C^\infty(\overline{\Omega})$ is continuous with respect to the supremum norm.

More generally, given a domain Ω with piecewise smooth boundary $\partial\Omega$, an associated Green's function can be constructed by finding a solution $h(x)$ to the system:

$$\begin{cases} \Delta h \ = \ 0 & \text{in} \quad \Omega \\ h \ = \ -\Gamma & \text{on} \quad \partial\Omega \end{cases} \tag{2.8}$$

and setting $G(x,y) \ = \ \Gamma(x-y) + h(x-y)$. Repeating the argument above, we arrive at analogous estimates for the continuity of the new solution mapping $\mathcal{G}$ for the domain Ω. In this general case, however, the explicit Green's function is no longer available (with the exception of a select number of simple geometries).

Although an explicit Green's function is not available for general domains, the maximum principle can still be applied since the constant C in Equation (2.7) depends only on the diameter of the domain Ω. In particular, when the diameters of the domains in consideration are bounded by a fixed constant, the maximum principle gives a uniform, domain-independent bound for the continuity of the solution mapping.

2.2.2 Discretization

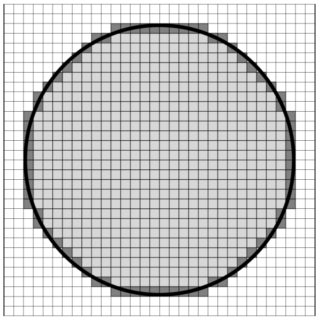


Figure 2.1. Discretization of the domain Ω (light gray) and boundary $\partial\Omega$ (dark gray) for the Poisson equation on the circle. The interior and boundary locations are used to define the network loss function and tailor training to the domain in consideration.

To cast the problem in discrete form for numerical approximation, we introduce a regular, rectangular grid Λ which covers the closure $\overline{\Omega}$ of each domain Ω in consideration. For our purposes, we consider domains contained inside the square $[-1,1]^{\times 2}$ and take Λ to be a uniform grid on the square with fixed resolution R. Given a domain Ω, we denote by Ω_d the collection of mesh points in Λ which lie inside of the closure of the domain Ω (i.e. $\Omega_d = \Lambda \cap \overline{\Omega}$). We then define the associated interpolation and projection mappings:

$$\mathcal{I} \,:\, \Big\{\, (x,\, f(x)) \,\Big\}_{x\,\in\,\Omega_d} \;\mapsto\; \text{interp}_{\overline{\Omega}}\Big\{(x,\, f(x))\Big\} \tag{2.9}$$

$$\mathcal{P} \,:\; u\,|_{\overline{\Omega}} \;\mapsto\; \Big\{\, (x,\, u(x)) \,\Big\}_{x\,\in\,\Omega_d} \tag{2.10}$$

where $\text{interp}_{\overline{\Omega}}\Big\{(x,\, f(x))\Big\}$ denotes the function defined on $\overline{\Omega}$ resulting from a fixed interpolation procedure (e.g. bilinear or bicubic interpolation). By construction, these mappings satisfy the norm estimates:

$$\sup_{\Omega} \left| \mathcal{I}\Big\{(x,\, f(x))\Big\} \right| \;\leq\; C(\text{interp}) \cdot \max\{|f(x)|\} \tag{2.11}$$

$$\max\Big\{\, |u(x)| \,:\, (x,\, u(x)) \,\in\, \mathcal{P}u \Big\} \;\leq\; \sup_{\Omega} |u| \tag{2.12}$$

where the constant $C(\text{interp})$ depends only on the fixed interpolation procedure which has been selected. It follows that the discretized, composite mapping:

$$\Big\{(x,\, f(x))\Big\}_{x\in\Omega_d} \xrightarrow{\;\mathcal{I}\;} f\,|_{\overline{\Omega}} \xrightarrow{\;\mathcal{G}\;} u\,|_{\overline{\Omega}} \xrightarrow{\;\mathcal{P}\;} \Big\{(x,\, u(x))\Big\}_{x\in\Omega_d} \tag{2.13}$$

is continuous and can, therefore, be approximated by a multi-layer feedforward neural network according to the *universal approximation theorem* [14, 30, 31]. In light of this, we pursue the construction of an approximate PDE solver using a neural network designed to predict discrete arrays $\{(x,\, u(x))\}$ of the solution point values on a grid provided a discrete array $\{(x,\, f(x))\}$ of specified source term values as input.

2.2.3 Approximation by convolutional networks

Our aim is to define a neural network which approximates the discretized solution mapping from Equation (2.13). The convolutional form of the integral operator $\mathcal{G}$ naturally inspires the use of convolutional layers within this neural network approximation. As outlined in Section 2.2, the task is rather trivial in the setting where the domain Ω in Equation (2.1) is fixed to the unit disk D. In this case, the true Green's function is known and can be used to directly apply the linear solution operator in Equation (2.5). For more general domains, however, the explicit Green's function is unavailable and alternative numerical methods must be employed. We first consider the problem setup on the circle as a proof-of-concept for the proposed neural network PDE solver framework; after establishing the performance of the network for this simple setup, we proceed to a more careful analysis of the following more complex scenarios:

1. *Varying domain:* construct a network to approximate the solution u to Equation (2.1) when both the source term f and domain Ω are permitted to vary.

2. *Nonlinear Poisson:* construct a network to approximate the solution u when both the source term f and domain Ω are permitted to vary, and Equation (2.1) is replaced with the following nonlinear PDE:

$$\begin{cases} \operatorname{div}\Big(\big(1+|u|^2\big)\cdot\nabla u\Big) \;=\; f & \text{in} \quad \Omega \\ u \;=\; 0 & \text{on} \quad \partial\Omega \end{cases} \tag{2.14}$$

The proposed framework can also be naturally extended to work with more general PDE systems. In particular, variable coefficient differential operators and homogeneous Neumann boundary conditions can both be accounted for with minimal changes to the network architecture.

2.3 Methodology

2.3.1 Bayesian framework

For the training procedure and theoretical foundation of the proposed model, we adopt the probabilistic framework introduced by Kingma and Welling [37]. In particular, we consider a setting in which two variational autoencoder (VAE) models have been trained: one for the space of source terms and one for the space of solutions. In this setup, the source data space $\mathcal{D}_f$ and solution data space $\mathcal{D}_u$ are mapped to associated latent variable spaces $\mathcal{Z}_f$ and $\mathcal{Z}_u$ by recognition models $p(z_f|f)$ and $p(z_u|u)$, respectively. The encoded variables are then decoded back to the original data spaces by generative models $q(f|z_f)$ and $q(u|z_u)$. The associated theoretical framework is summarized in the commutative diagram provided in Figure 2.2.

The mapping $\mathcal{G}_z$ corresponds to the latent space transformation which sends the encoded latent variables z_f associated with the source term f to the latent variables z_u corresponding to the solution u.

The proposed neural network model incorporates a recognition model $p(z_u|f)$, trained to map functions from the source data space $\mathcal{D}_f$ directly to the solution representations z_u, along with a generative model $q(u|z_u)$, designed to produce approximations to the true

$$\begin{tikzcd}
\boxed{\mathcal{D}_f} \arrow[r, "p(z_f|f)"] \arrow[dr, dashed, "p(z_u|f)"'] \arrow[dr, phantom, "\circ", shift left=3] & \boxed{\mathcal{Z}_f} \arrow[r, "q(f|z_f)"] \arrow[d, dashed, "\mathcal{G}_z"] \arrow[dr, phantom, "\circ"] & \boxed{\mathcal{D}_f} \arrow[d, "\mathcal{G}"] \\
\boxed{\mathcal{D}_u} \arrow[r, "p(z_u|u)"'] & \boxed{\mathcal{Z}_u} \arrow[r, "q(u|z_u)"'] & \boxed{\mathcal{D}_u}
\end{tikzcd}$$

Figure 2.2. Commutative diagram for the proposed recognition/generator model; the source data space $\mathcal{D}_f$ and solution data space $\mathcal{D}_u$ are mapped to associated latent variable spaces $\mathcal{Z}_f$ and $\mathcal{Z}_u$ by recognition models $p(z_f|f)$ and $p(z_u|u)$, respectively. The encoded variables are then decoded back to the original data spaces by generative models $q(f|z_f)$ and $q(u|z_u)$.

solutions from the encoded representations. This has the benefit of resulting in a one-shot training process (i.e. separate training procedures are not required for each VAE model) as well as enabling the learned latent space structure to be optimized simultaneously for the data space encoder and solution space decoder.

2.3.2 Probabilistic training procedure

One shortcoming of traditional numerical methods is the lack of any explicit uncertainty quantification associated with the resulting approximations. Though rigorously grounded in asymptotic convergence results, the uncertainty for each individual approximation is known only up to a single, global bound. The approximate solutions produced by a FEM solver, for example, do not provide any indication of specific regions in a domain where the convergence/accuracy of the numerical solution is uncertain. Conventional neural network architectures follow this format as well, predicting pointwise values without providing estimates gauging how far off the prediction may be from the true solution in certain regions.

To address this issue, we construct a network with the capacity to quantify the *heteroscedastic* uncertainty associated with its predictions; that is, the uncertainty specific to individual examples (e.g. the uncertainty resulting from a sharp corner in a given domain). As noted by [35, 49], this can be achieved by casting the network's predictions in the form of Gaussian posterior distributions, as opposed to singular point-estimates. The network is designed to predict pointwise statistics $\widehat{\mu}[\mathrm{i},\mathrm{j}]$ and $\widehat{\sigma}[\mathrm{i},\mathrm{j}]$ corresponding to a posterior distribution $\mathcal{N}(\widehat{\mu}[\mathrm{i},\mathrm{j}],\widehat{\sigma}[\mathrm{i},\mathrm{j}])$ over the possible true solution values $\{u(x_\mathrm{i},y_\mathrm{j})\}$. This allows the network to attach to each point both an estimate $\widehat{\mu}[\mathrm{i},\mathrm{j}]$ for the value of the solution and a standard deviation $\widehat{\sigma}[\mathrm{i},\mathrm{j}]$ reflective of its confidence in that estimate. For numerical stability, and to avoid enforcing constraints on the network's variance predictions, it is convenient to predict the log standard deviations $\widehat{\sigma}_{\log}$ and compute the associated statistics for the loss function using the exponential function:

$$\widehat{\sigma}[\mathrm{i},\mathrm{j}] \;=\; \exp\Big(\widehat{\sigma}_{\log}[\mathrm{i},\mathrm{j}]\Big) \tag{2.15}$$

Training is then designed to maximize the associated likelihood of observing the known true solution values. Assuming, for example, independent pointwise posterior distributions, we aim to maximize:

$$p(u\,;\,\widehat{\mu},\widehat{\sigma}) \;=\; \prod_{\mathrm{i},\mathrm{j}=1}^{R} \frac{1}{\sqrt{2\pi\cdot\widehat{\sigma}[\mathrm{i},\mathrm{j}]^2}}\,\exp\bigg(-\tfrac{1}{2}\Big(u[\mathrm{i},\mathrm{j}]-\widehat{\mu}[\mathrm{i},\mathrm{j}]\Big)^2\big/\,\widehat{\sigma}[\mathrm{i},\mathrm{j}]^2\bigg) \tag{2.16}$$

or, equivalently, to minimize the negative log-likelihood of the observed solution values:

$$-\log p(u\,;\,\widehat{\mu},\widehat{\sigma}) \;=\; \sum_{\mathrm{i},\mathrm{j}=1}^{R}\tfrac{1}{2}\Big(u[\mathrm{i},\mathrm{j}]-\widehat{\mu}[\mathrm{i},\mathrm{j}]\Big)^2\big/\,\widehat{\sigma}[\mathrm{i},\mathrm{j}]^2 \,+\, \sum_{\mathrm{i},\mathrm{j}=1}^{R}\tfrac{1}{2}\,\log(2\pi\cdot\widehat{\sigma}[\mathrm{i},\mathrm{j}]^2) \tag{2.17}$$

This framework allows the network to begin by making coarse mean estimates while admitting relatively high variance estimates (i.e. low-precision predictions) and to gradually increase the network's predicted confidence by lowering variance estimates as the training procedure progresses. This interpretation is indeed motivated by the design of the network's loss function, which is fundamentally responsible for guiding the training process, as illustrated in Figure 2.3. More importantly, the interpretation is seen to coincide with what happens in practice, as will be discussed further in Section 2.4.

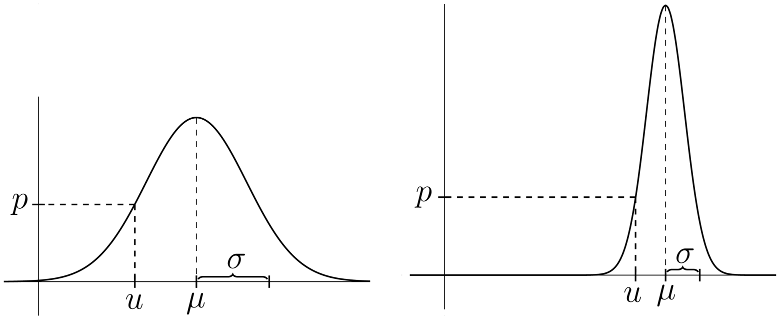


Figure 2.3. The probabilistic prediction framework allows the network to begin with coarse, low-confidence predictions (left) and to gradually build confidence by lowering the predicted standard deviations. High confidence predictions (right) allow the network to attain far better losses when correct, but have steep drop-offs penalizing any inaccuracy in the prediction.

2.3.3 Network architecture

The proposed network architecture consists of two primary components; an encoder designed to map high-level input functions to low-dimensional latent features, and a decoder used to map these latent features to approximate solutions. The encoder consists of a series of convolutional layers which gradually reduce the resolution of input features. The output features of the encoder are then partitioned into mean and standard deviation values and used to sample latent features. The standard 'reparameterization trick' [37] is used to maintain a differentiable network structure; the entries of the latent features are sampled independently via $z = \mu + \sigma \cdot \varepsilon$ where μ and σ are the corresponding entries of the reshaped encoded features and $\varepsilon \sim \mathcal{N}(0, 1)$.

The decoder maps the sampled latent features back to the original resolution using a series of convolutional layers followed by bilinear interpolation. In addition, the network incorporates aspects of the U-Net architecture [74], passing features extracted in the encoding component directly over to the decoder. In particular, the encoder features with spatial resolutions 32×32, 16×16, and 8×8 are concatenated with the features of the same resolution in the network's decoder (as indicated with arrows in Figure 2.4). To accommodate the probabilistic prediction framework described in Section 2.3.2, the stem of the decoder is

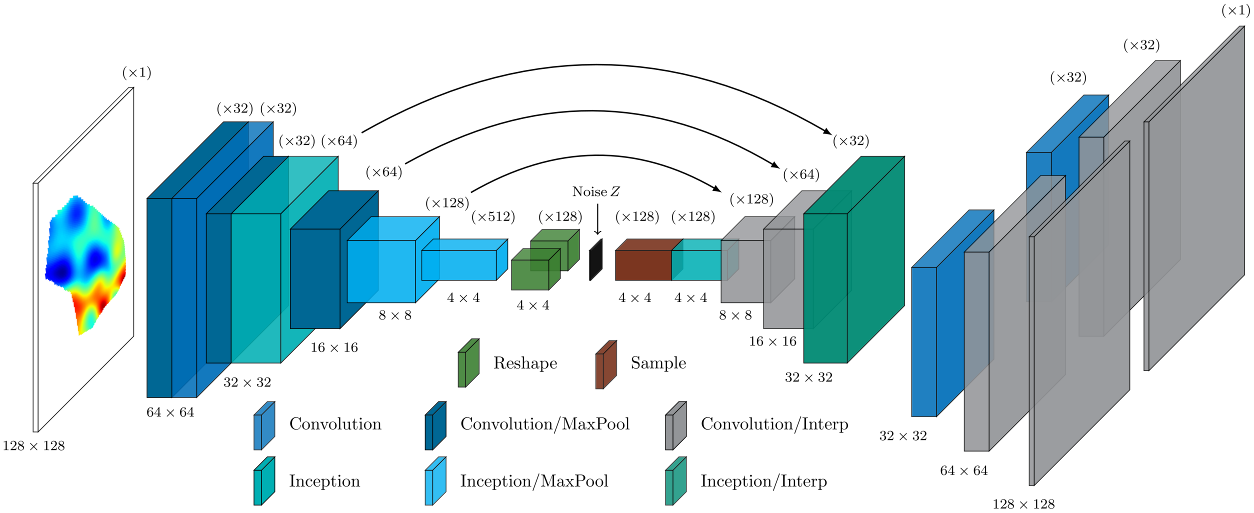


Figure 2.4. Proposed network architecture. Features have been color-coded according to the type of layer which has been used to produce them. The arrows indicate feature concatenation, whereby the encoder features are shared/reused in the decoding process in accordance with the U-Net architecture. For the probabilistic network, the stem of the decoder is split into two branches corresponding to mean and variance predictions (as shown above); the MSE network architecture is obtained by simply dropping the variance branch of the decoder (since a variance prediction is not required in this case).

split into two branches: one for the mean predictions and another for log standard deviation predictions.

For improved performance, some layers have been split into a collection of collaborative filters [77] which are referred to as 'inception blocks' in Figure 2.4 for convenience. These blocks consist of a max-pooling layer along with 1×1, 3×3, and two stacked 3×3 convolutional layers implemented in parallel; the features of these layers are then concatenated channel-wise to produce the final set of features sent to the next network layer. Dropout layers with drop-rate 0.045 have also been included before and after the first inception block in the decoder to help avoid over-fitting to the training data set.

2.3.4 Network loss functions and training procedure

Based on the problem statement provided in Equation (2.1), it is natural to define the network's loss function in terms of two components; one to enforce the differential equation on the interior, and another to enforce the boundary condition. To this end, the mean squared error (MSE) can be calculated on the interior and the boundary of the domain using a template file to indicate where these errors should be calculated:

$$\mathrm{Loss}_{\texttt{MSE}}(\,\widehat{u}\,;\,u,\Omega) \;=\; \frac{1}{|\Omega|}\sum_{\mathrm{i,j}=1}^{R} \mathbb{1}_{\Omega}[\mathrm{i,j}]\cdot\Big(\,\widehat{u}[\mathrm{i,j}] - u[\mathrm{i,j}]\,\Big)^2 + \frac{\lambda}{|\partial\Omega|}\sum_{\mathrm{i,j}=1}^{R} \mathbb{1}_{\partial\Omega}[\mathrm{i,j}]\cdot\Big(\,\widehat{u}[\mathrm{i,j}] - u[\mathrm{i,j}]\,\Big)^2 \tag{2.18}$$

Here R denotes the selected output resolution, $|D|$ corresponds to the number of pixels inside the region D, and $\mathbb{1}_D$ denotes the indicator function for the discretized, Boolean template file corresponding to the domain D (e.g. $\mathbb{1}_{\Omega}[\mathrm{i,j}] = 1$ when the $(\mathrm{i,j})^{th}$ pixel lies inside of the domain Ω, and $\mathbb{1}_{\Omega}[\mathrm{i,j}] = 0$ otherwise).

The hyperparameter λ in Equation (2.18) is intended to balance the contributions of the interior and boundary terms. In practice, however, it has been observed that this explicit boundary loss term is not strictly necessary. Setting $\lambda = 0$, we arrive at the alternative, simplified expression for the MSE loss:

$$\mathrm{Loss}_{\texttt{MSE}}(\,\widehat{u}\,;\,u,\Omega) \;=\; \frac{1}{|\Omega|}\sum_{\mathrm{i,j}=1}^{R} \mathbb{1}_{\Omega}[\mathrm{i,j}]\cdot\Big(\,\widehat{u}[\mathrm{i,j}] \;-\; u[\mathrm{i,j}]\,\Big)^2 \tag{2.19}$$

The probabilistic training procedure described in Section 2.3.2, and illustrated in Figure 2.5, is implemented by replacing the traditional MSE loss function with the following negative log-likelihood calculation:

$$\mathrm{Loss}_{\texttt{PROB}}(\,\widehat{\mu},\widehat{\sigma}\,;\,u,\Omega) \;=\; \frac{1}{|\Omega|}\sum_{\mathrm{i,j}=1}^{R} \mathbb{1}_{\Omega}[\mathrm{i,j}]\cdot\Big[\,\tfrac{1}{2}\Big(\,\widehat{\mu}[\mathrm{i,j}] \;-\; u[\mathrm{i,j}]\,\Big)^2\,/\,\widehat{\sigma}[\mathrm{i,j}]^2 \;+\; \tfrac{1}{2}\,\log(2\boldsymbol{\pi}\cdot\widehat{\sigma}[\mathrm{i,j}]^2)\,\Big] \tag{2.20}$$

Intuitively, the two terms of this summand can be seen to account for a notion of *model fitness* and *model confidence*, respectively; the training procedure is thus directed toward balancing the trade-offs between the two competing notions in order to minimize the probabilistic loss function.

In addition to the primary MSE/probabilistic loss term, the network incorporates a secondary loss term corresponding to the Kullback-Leibler (KL) divergence prescribed by

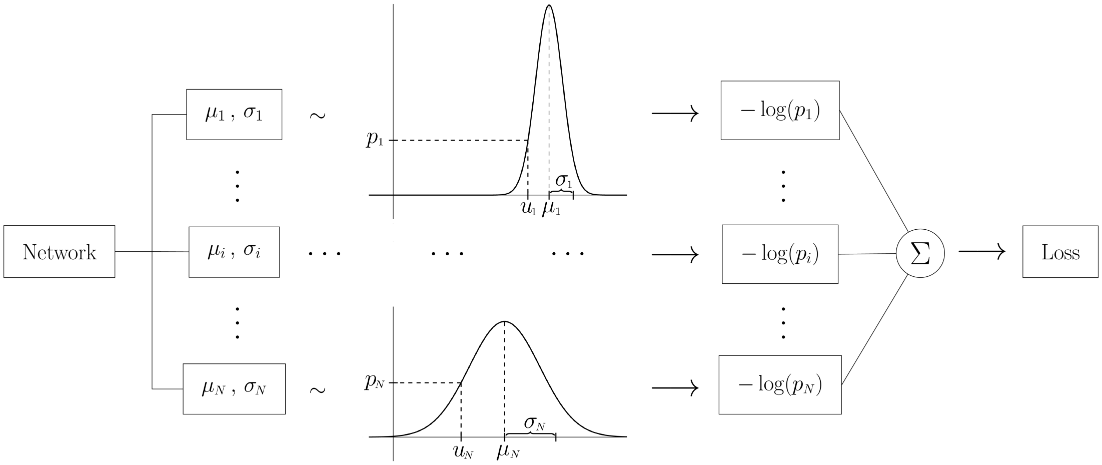


Figure 2.5. Overview of the proposed probabilistic network design; point estimates $\widehat{u}_\mathrm{i}$ are replaced with the statistics μ_i and σ_i of a normal distribution, and the negative log-likelihoods of observed solution values u_i are summed to define the network loss. To align with the numerical implementation, we have flattened the summation operation in Equation (2.20) and omitted terms in which $\mathbb{1}_\Omega[\mathrm{i},\mathrm{j}] = 0$ so that only the $N = |\Omega|$ points inside of the current domain are included in the loss calculation.

the VAE framework. As noted in [37], assuming independent normal priors on the latent means z_μ and variances z_{σ^2}, this can be computed via:

$$\mathrm{Loss}_{\mathtt{KL}}(z_\mu, z_\sigma) \;=\; \frac{1}{2}\sum_{\mathrm{i}=1}^{L}\Big(\, z_{\mu,\mathrm{i}}^2 \,+\, z_{\sigma^2,\mathrm{i}} \,-\, \log z_{\sigma^2,\mathrm{i}} \,-\, 1\Big) \tag{2.21}$$

where L denotes the number of latent variables at the sampling stage of the model, and $z_{\mu,\mathrm{i}}$ and $z_{\sigma^2,\mathrm{i}}$ denote the encoded mean and variance associated with the ith latent variable. In particular, for the experiments presented in this work, there are $L = 2048$ latent variables corresponding to the individual entries of the 128 channels of 4×4 features at the bottleneck of the network architecture, as illustrated in Figure 2.4.

Once the loss function for the network is specified, training is carried out using the Adam optimization algorithm [36] to minimize the loss function evaluated on batches of data from the training set. A batch size of 64 has been used, and the learning rate of the Adam optimizer has been initialized at 0.00075, with an exponential decay applied every $10,000$ steps by a factor of 0.95. The network architecture and hyperparameters have been tuned to optimize the MSE training procedure. The same architecture and hyperparameters were observed to be optimal for the probabilistic training procedure as well (with the final layers of the decoding component duplicated to produce uncertainty estimates), and this same network setup is shown to be suitable for each of the three problem setups into consideration.

2.3.5 Data generation

The source terms for each experiment are taken to be samples from mean-zero Gaussian random fields:

$$f \;\sim\; \mathcal{G}(0, k_l(x,y)) \tag{2.22}$$

where the covariance kernels $k_l(x,y) = \exp(-\|x-y\|^2/2l^2)$ correspond to squared exponential kernels with *length-scales* l varying in a fixed interval $[l_{min}, l_{max}]$. In particular, 20 distinct length-scales were selected between $l_{min} = 0.2$ and $l_{max} = 0.6$ were selected for data creation in the experiments presented in this work. We note that this setup explicitly defines the universal data distribution, in accordance with the proposed Bayesian framework for training described in Section 2.3.1. In particular, the data samples for the source terms f are drawn from a distribution of functions in $C^\infty(\Omega)$ with an average expected rate of change controlled by the length-scale parameters l in the specified collection of Gaussian random fields.

The domain generation procedure has been designed to sample from a family of polygonal domains in the interior of the region $[-1,1]^{\times 2} \subset \mathbb{R}^2$ with vertex counts between $v_{min} = 4$ and $v_{max} = 16$. This generation was carried out by sampling a vertex count $v \sim \text{Unif}(\{v_{min}, \ldots, v_{max}\})$, randomly sampling angles for vertices to be placed at $\{\theta_k\} \sim \text{Unif}(0, 2\pi)$, and lastly sampling the corresponding radii for the vertices $\{r_k\} \sim \text{Unif}(r_{min}, 1)$ with $r_{min} = 0.25$. After re-indexing to ensure $\theta_1 < \cdots < \theta_v$, the boundary of the generated domain was taken to correspond to the cycle $[(r_1\cos\theta_1, r_1\sin\theta_1), \ldots, (r_v\cos\theta_v, r_v\sin\theta_v)]$.

Once the source terms and domains have been generated, it remains only to produce the corresponding solution to each system. The approximate solutions $\{u(x)\}$ used for the training datasets have been computed numerically using the finite element solver FEniCS with a mesh resolution of 35 (corresponding to finite element cells of approximate diameter $1/35 \approx 0.0286$) and Lagrange basis functions of polynomial order 1. The generated source terms and meshes have also been implemented within FEniCS to facilitate the solution procedure; the final source terms, meshes, and solutions are then evaluated on a regular grid and converted to rectangular arrays to accommodate for a more natural neural network architecture. Of note is the fact that the data generation process outlined above is trivially parallelizable, as the solutions to distinct systems are independent. This fact has indeed been leveraged in the code provided to take full advantage of all processors available, and can further be implemented on distributed systems with minimal changes.

The numerical results presented in this work have been obtained using datasets consisting of $100,000$ examples, with 80% used for training and 20% used for validation. These datasets have been generated following the procedures above with $5,000$ source term samples drawn from each of the 20 length-scale classes. The mesh, source term, and solution arrays are also rotated 180 degrees and flipped horizontally during training to augment the effective dataset size to $400,000$ examples for each problem setup.

2.4 Numerical results

2.4.1 Comparison of training procedures

Following the methodology outlined in Section 2.3, a convolutional network has been trained to serve as a numerical solver for each of the three problem setups. The networks have each been trained independently using the traditional MSE loss as well as the proposed probabilistic training procedure. As shown in Table 2.1, the probabilistic framework outperforms the conventional MSE training in each of the three problem setups.

This is a rather striking result considering the fact that the mean estimates of the probabilistic networks (which are used to calculate the errors in Table 2.1) are produced using an identical architecture to those used for the MSE networks. Introducing an independent variance branch in the decoder and redefining the loss function in terms of likelihood estimates is actually seen to achieve lower L^2 errors, the precise errors the MSE network is specifically designed to minimize. The consistent outperformance of the probabilistic network is also

Table 2.1. Summary of the network errors for each of the three problem setups under consideration. The L^1 relative errors and mean squared errors are provided for both the training dataset (left) and validation dataset (right). The relative errors are seen to gradually increase as we move from the simple setup on the circle to the more complex nonlinear equation on varying domains. In all cases, however, we see that the probabilistic training procedure outperforms the traditional mean squared error training.

Problem Setup	Model	L^1 Relative Error		Mean Squared Error	
Poisson on Circle	**Probability**	**9.19e−3**	**1.00e−2**	**1.18e−4**	**1.50e−4**
	MSE ($\lambda = 0.1$)	1.23e−2	1.28e−2	2.60e−4	3.06e−4
	MSE ($\lambda = 0.0$)	1.23e−2	1.29e−2	2.48e−4	2.90e−4
Varying Domain	**Probability**	**1.82e−2**	**2.11e−2**	**1.21e−3**	**1.45e−3**
	MSE ($\lambda = 0.1$)	3.43e−2	3.57e−2	2.25e−3	2.62e−3
	MSE ($\lambda = 0.0$)	3.60e−2	3.75e−2	2.43e−3	2.86e−3
Nonlinear Poisson	**Probability**	**1.94e−2**	**2.24e−2**	**1.32e−3**	**1.58e−3**
	MSE ($\lambda = 0.1$)	3.21e−2	3.46e−2	1.84e−3	2.46e−3
	MSE ($\lambda = 0.0$)	3.37e−2	3.61e−2	2.09e−3	2.69e−3

observed when the data set is partitioned according to the length-scales of the input functions, and, as one would expect, both networks admit a gradual reduction in accuracy as the length-scales of the inputs decrease, corresponding to more oscillatory source terms.

In addition to the unexpected improvement in performance, the probabilistic networks have the added benefit of providing predictive uncertainties associated with their predictions. This gives the probabilistic networks the capacity to clearly indicate potential inaccuracies in its predictions and is seen to occur in practice as shown in Figure 2.6. Of note is the fact that these examples have been taken from the validation data set, and have therefore never been seen by the network during the training procedure. The fact that the uncertainty estimates remain accurate for the validation problems suggests that the network has learned an uncertainty quantification schema capable of generalization, and is not simply fit to model the observed errors during training. A closer, more quantitative analysis of these uncertainties is provided below.

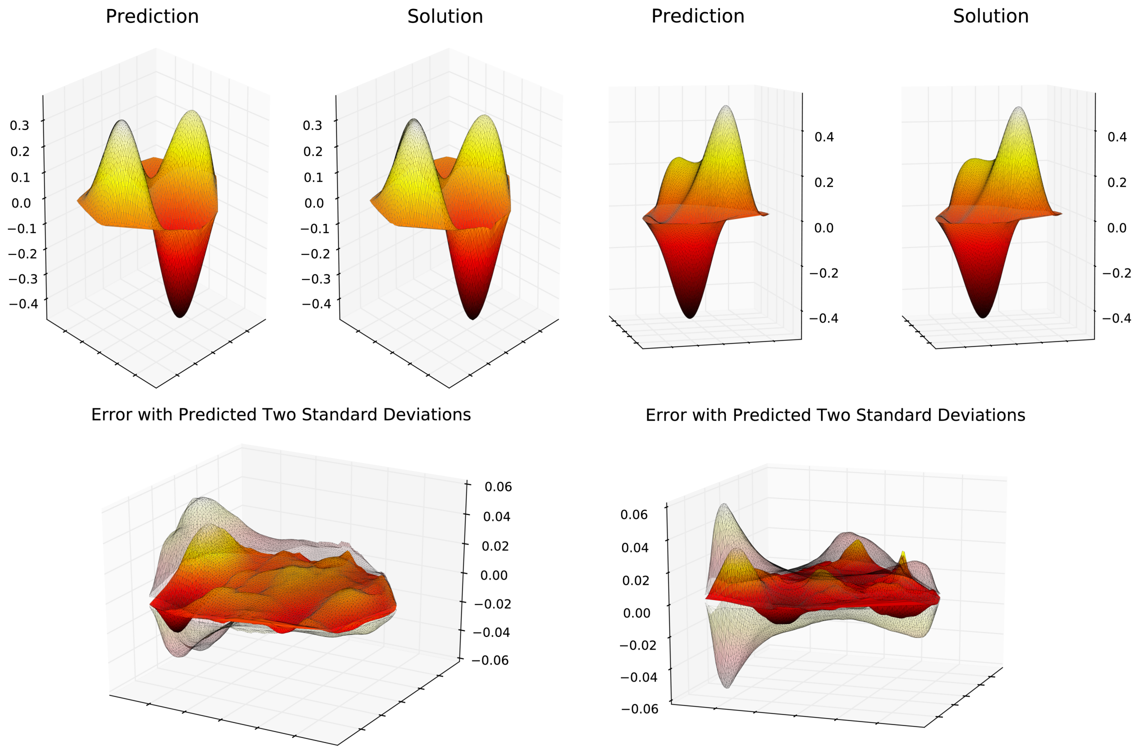


Figure 2.6. Qualitative results demonstrating the network's predictions for two problems in the "Varying Domain Poisson Equation" setup. Both examples have been taken from the validation data set and have not been seen by the network during training. Network predictions and true solutions are shown at the top, with the corresponding absolute errors plotted below along with the network's predicted two standard deviations plus/minus bounds (shown as transparent wireframes). The predicted pointwise standard deviations are seen to provide accurate error estimates for the network predictions.

2.4.2 Uncertainty quantification

An essential requirement of any meaningful measure of uncertainty is that the predicted uncertainties are correlated with the associated prediction errors. As shown in Figure 2.7, this is indeed the case for the proposed framework; the predicted uncertainties are seen to gradually reduce throughout the training process, closely following the progression of the network's mean squared error on the validation data set. A strong relationship between the model uncertainty and the absolute L^2 error is observed for the full duration of training, with a Pearson correlation coefficient of 0.921.

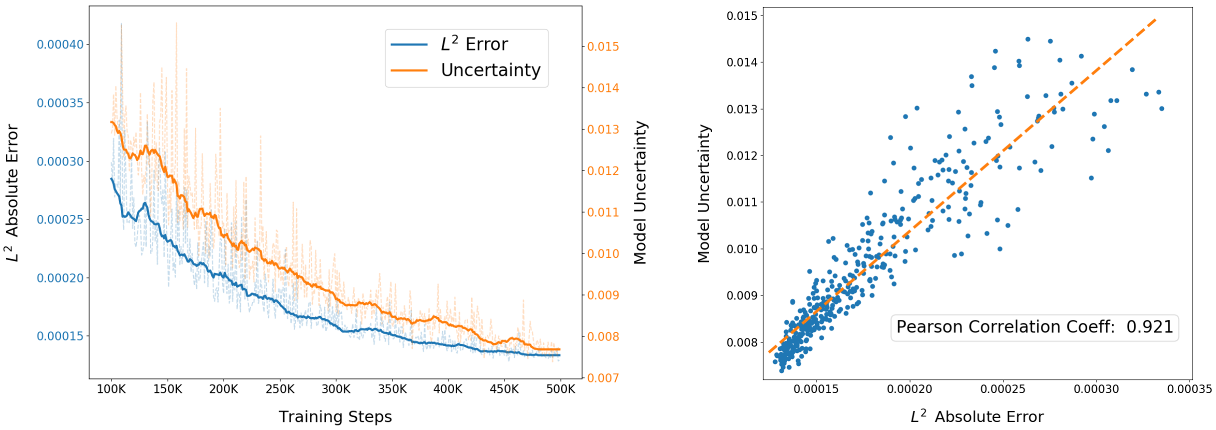


Figure 2.7. The absolute L^2 loss and magnitude of predicted uncertainties in the "Varying Domain Poisson Equation" setup throughout the training process (left) along with a plot of the absolute L^2 error against the predicted model uncertainty (right). A high correlation between the predicted model uncertainty and error in network predictions is clearly visible, with a Pearson correlation coefficient of 0.921.

By construction, the trained network is designed to predict pointwise Gaussian posterior distributions for the true solution values. To evaluate how effectively the network conforms to this design in practice, we compare the trained network predictions with true solutions. We should expect the fraction of pixels falling within a specified standard deviation range to coincide with that of a Gaussian distribution:

$$\frac{1}{|\mathcal{D}|}\sum_{d\in\mathcal{D}}\sum_{[\mathrm{i,j}]\in\Omega_d}\frac{1}{|\Omega_d|}\,\mathbb{1}\Big(\big|\widehat{\mu}_d[\mathrm{i,j}]\;-\;u_d[\mathrm{i,j}]\big|\;\le\;x\cdot\widehat{\sigma}_d[\mathrm{i,j}]\Big)\;\approx\;\mathbb{P}\Big(|\mathcal{N}(0,1)|\;\le\;x\Big) \tag{2.23}$$

Here the left-hand-side corresponds to the average number of points where the true solution differs from the predicted mean by less than a factor x of the predicted standard deviation:

$$\text{i.e.}\quad \widehat{\mu}_d[\mathrm{i,j}]\;-\;x\cdot\widehat{\sigma}_d[\mathrm{i,j}]\;\le\;u_d[\mathrm{i,j}]\;\le\;\widehat{\mu}_d[\mathrm{i,j}]\;+\;x\cdot\widehat{\sigma}_d[\mathrm{i,j}]\qquad d\in\mathcal{D}\,,\;\;[\mathrm{i,j}]\in\Omega_d \tag{2.24}$$

As shown in Figure 2.8, the trained model follows the Gaussian design almost perfectly in practice. In particular, the observed errors between the network predictions and true solutions are seen to closely approximate the empirical 68-95-99.7 rule for normal distributions.

This experimental evidence suggests that the network's predicted error bounds do, in fact, provide an accurate measure of the model's uncertainty.

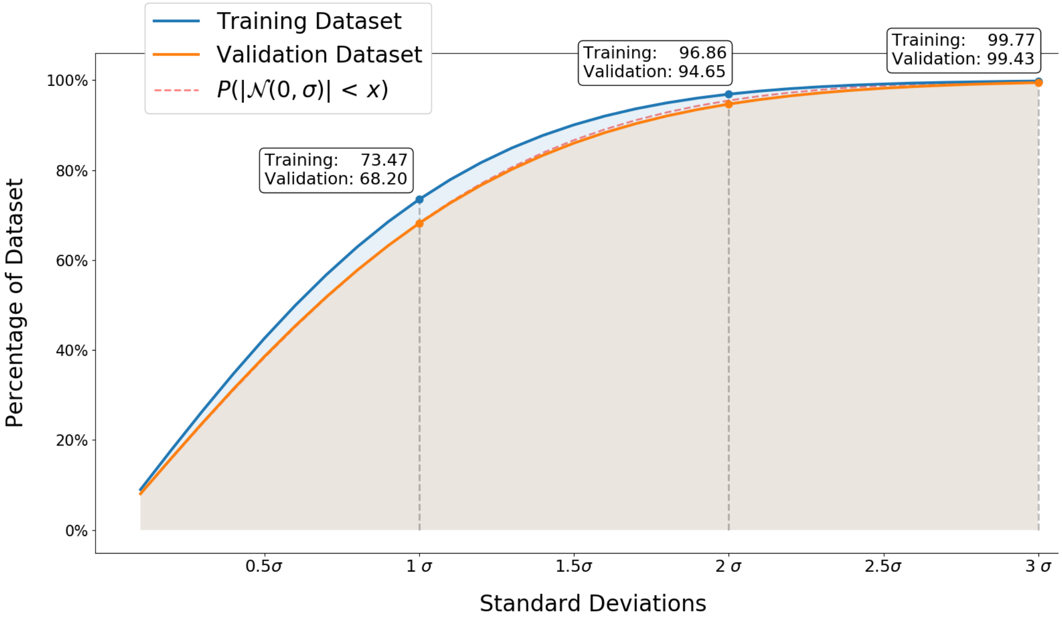


Figure 2.8. Quantitative analysis of the network's predicted standard deviation values in the "Varying Domain Poisson Equation" setup. The percentage of points where the difference between the network prediction and true solution is less than a multiple of the predicted pointwise standard deviation values is shown for the training (top curve) and validation (bottom curve) data sets. The curve for the validation data set is shown to closely follow the cumulative distribution of the half-normal distribution (dashed, partially covered by validation), supporting the claim that the predicted standard deviation values do, in fact, provide accurate pointwise uncertainty estimates.

2.4.3 Inference speed

The numerical results for each experiment have been obtained using an 8-core Intel Xeon 3.60GHz processor and a single NVIDIA GeForce GTX 1080 GPU. To compare the inference/prediction speed of the proposed convolutional networks with the traditional FEM approach, the average inference times have been evaluated for solving $10,000$ distinct PDE systems. As shown in Table 2.2, the convolutional networks achieve faster inference times for each problem setup, with a speed-up by a factor of 3.1 using the CPU alone and a speed-up by a factor of 21.3 using a single GPU. For a fair comparison with the batch evaluation of the convolutional networks, the FEM implementation in FEniCS has been parallelized to use all 8 cores.

Table 2.2. Comparison of inference/prediction times for the convolutional and FEM solvers. The "FEniCS (Coarse)" results correspond to FEM implementations in which the meshes have been coarsened until the accuracy coincides with that of the neural network models. The accuracies of the coarse FEM implementation and neural network models have been measured with respect to the refined mesh "FEniCS" results which serve as proxies for the true solutions.

Problem Setup	FEniCS	FEniCS (Coarse)	Network	Network (GPU)
Poisson on Circle	0.05064 seconds	0.03872 seconds	0.01157 seconds	0.00171 seconds
Varying Domain	0.04432 seconds	0.03395 seconds	0.01160 seconds	0.00170 seconds
Nonlinear Poisson	0.04863 seconds	0.03473 seconds	0.01152 seconds	0.00164 seconds
Average	0.04786 seconds	0.03580 seconds	0.01156 seconds	0.00168 seconds

2.4.4 Concluding remarks

In this chapter, we have demonstrated a framework for the construction of approximate PDE solvers on varied two-dimensional domains by leveraging existing numerical methods and recent advances in data-driven deep learning. A theoretical justification for the use of neural networks as approximations to PDE solution mappings on varied domains is established using the theory of Green's functions and the universal approximation property of neural networks. This framework replaces online tasks such as mesh generation, finite element space construction, and linear/nonlinear system solvers with a one-shot, offline training session. After training, predictions can be obtained on any specified two-dimensional domain via a single forward-pass through a light-weight convolutional network. The networks are also designed to provide pointwise error bounds along with the approximate solutions; this is achieved using a simplified deterministic training procedure which avoids the computationally expensive inference steps of BNNs and allows training to be scaled up to work with the large data sets required for learning on varied geometries.

The performance of the framework has been demonstrated for both linear and nonlinear heterogeneous elliptic PDEs on varied two-dimensional domains with homogeneous Dirichlet boundary conditions. The proposed framework can also be naturally extended to handle inhomogeneous PDE coefficients, such as stiffness terms, by simply adding additional channels to the network's input. The experimental results presented in this chapter provide a foundation for the construction of more general neural network solvers in the future; it is envisioned that the long-term development of this framework will result in networks capable of solving linear and nonlinear heterogeneous PDEs with variable coefficients and inhomogeneous/mixed boundary conditions on arbitrary domains. To reach this goal, however, a significant amount of additional research efforts will need to be directed toward effectively modeling inhomogeneous and mixed boundary conditions. In the following chapter, we investigate an alternative neural network framework for approximating solution operators for PDEs which is capable of incorporating inhomogeneous boundary conditions, however the scope of these operators are currently restricted to fixed spatial domains.

2.A Additional experiments

2.A.1 Variable coefficient differential operators

The proposed framework provides a natural extension for handling variable coefficient differential operators as well. In particular, the coefficients of the operator can be passed to the network by simply concatenating the coefficient arrays with the source term array. Each channel of the network's input then corresponds to a specific coefficient or source term in the differential equation. For example, we may consider the elliptic differential equation:

$$\begin{cases} \operatorname{div}\Big(a \cdot \nabla u\Big) \;=\; f & \text{in} \quad \Omega \\ u \;=\; 0 & \text{on} \quad \partial\Omega \end{cases} \tag{2.25}$$

where the coefficient $a \in C^\infty(\overline{\Omega})$ is subject to the elliptic coercivity constraint $\inf_\Omega a > \kappa$ for some fixed $\kappa > 0$. The same randomization procedure used to generate the source terms f can be applied to sample coefficient terms a with one additional step designed to enforce the coercivity constraint: the values of each coefficient array are rescaled and shifted to have a mean value of 1.0 and satisfy $\inf_\Omega a > \kappa$ with $\kappa = 0.2$.

In this case, the network input consists of two channels: one for the source term f and another for the coefficient a. Conveniently, no additional modification to the network architecture is required; the first convolutional layer identifies the additional input channel, adds the necessary filters for parsing the channel, and passes the extracted features to subsequent hidden layers without further modification. The qualitative and quantitative results for the variable coefficient problem setup defined in Equation (2.25) are provided in Figure 2.9 and Table 2.3, respectively.

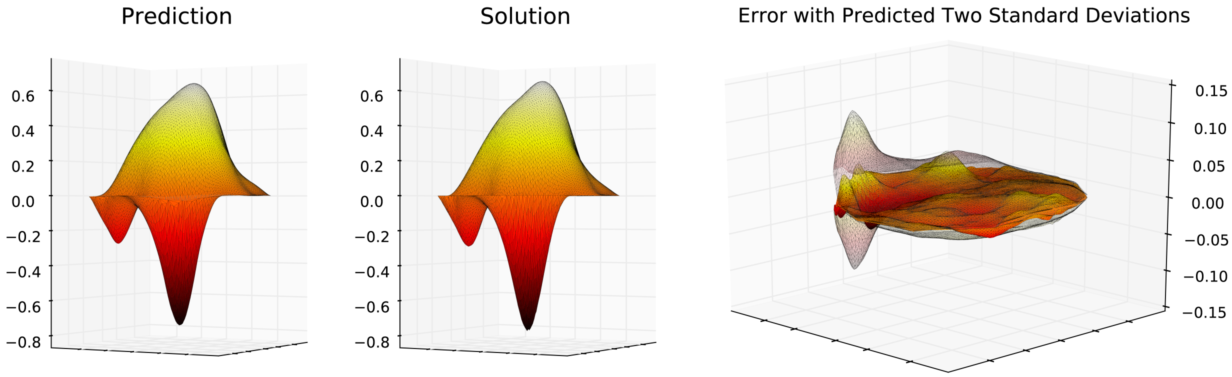


Figure 2.9. Qualitative results for the "Variable Coefficient" problem setup.

Table 2.3. Quantitative results for the "Variable Coefficient" problem setup.

Problem Setup	Model	L^1 Relative Error		Mean Squared Error	
Variable Coefficient	**Probability**	**2.06e−2**	**2.62e−2**	**7.95e−4**	**1.33e−3**
	MSE ($\lambda = 0.1$)	4.25e−2	4.54e−2	2.82e−3	3.42e−3
	MSE ($\lambda = 0.0$)	3.81e−2	4.19e−2	2.28e−3	3.00e−3

2.A.2 Neumann boundary conditions

The proposed framework is also compatible with homogeneous Neumann boundary conditions. In particular, denoting the outward unit normal vector along the boundary of the domain Ω by $\vec{n}$, we consider the following problem:

$$\left\{\begin{array}{rclcl} \Delta u & = & f & \text{in} & \Omega \\ \dfrac{\partial u}{\partial \vec{n}} & = & 0 & \text{on} & \partial\Omega \end{array}\right. \tag{2.26}$$

In this situation, no modification to the network architecture is required. The training dataset is simply modified to reflect the Neumann boundary conditions, and the network naturally adapts its predictions to approximate the target boundary behavior.

In practice, however, the network is observed to perform poorly on domains with sharp corners in the Neumann boundary condition setup. This is a rather natural difficulty since the outward normal derivative constraints at domain vertices include severe discontinuities. To minimize the impact this has on the network's training procedure, the dataset has been generated with an additional domain smoothing step intended to remove sharp corners from the randomized geometries.

The qualitative and quantitative results for the Neumann boundary condition setup defined in Equation (2.26) are provided in Figure 2.10 and Table 2.4, respectively.

Table 2.4. Quantitative results for the "Neumann" problem setup.

Problem Setup	Model	L^1 Relative Error		Mean Squared Error	
Neumann BC	**Probability**	**4.23e−2**	**4.64e−2**	**1.78e−2**	**1.84e−2**
	MSE ($\lambda = 0.1$)	1.29e−1	1.33e−1	3.18e−2	3.38e−2
	MSE ($\lambda = 0.0$)	1.31e−1	1.34e−1	3.26e−2	3.46e−2

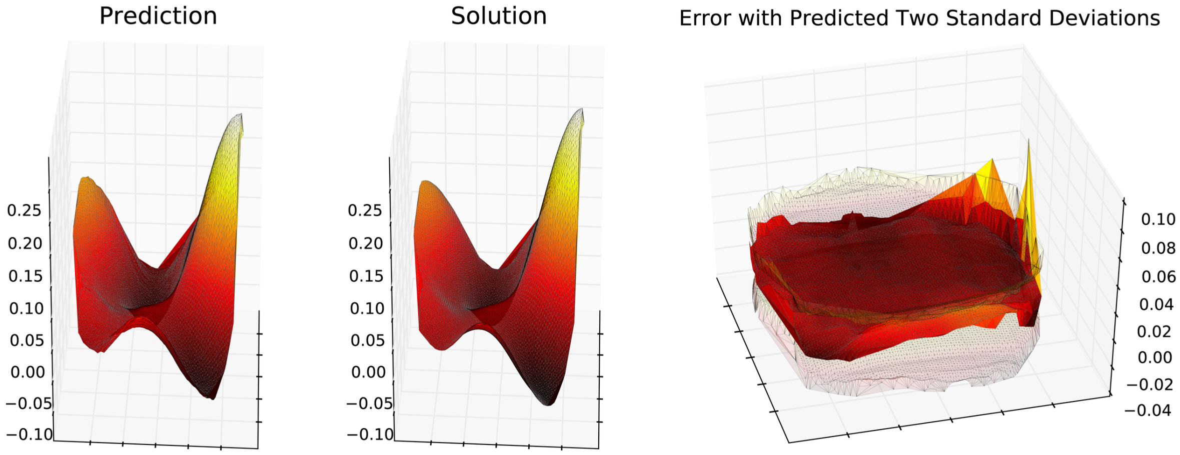


Figure 2.10. Qualitative results for the "Neumann" problem setup. The network's predictive uncertainty is observed to increase sharply near the boundary of the domain, forming a bowl-like shape. The boundary of the domain is also where the most severe inaccuracies in the network's predictions occur, and this information is reflected in the uncertainty estimates provided by the network.

2.B Alternative distributions for modeling network uncertainty

A natural extension to the proposed uncertainty schema can be obtained by replacing the normally distributed network predictions with an alternative distribution. For example, the likelihood associated with independent pointwise normal distributions:

$$p_{\text{Normal}}(u\,;\,\mu,\sigma) \;=\; \prod_{\mathrm{i,j}=1}^{R} \frac{1}{\sqrt{2\pi\cdot\sigma[\mathrm{i,j}]^2}}\, \exp\Big(\, -\tfrac{1}{2}\big(u[\mathrm{i,j}]-\mu[\mathrm{i,j}]\big)^2 / \,\sigma[\mathrm{i,j}]^2 \,\Big) \tag{2.27}$$

can be replaced with the likelihood associated with independent pointwise Laplace or Cauchy distributions:

$$p_{\text{Laplace}}(u\,;\,\mu,b) \;=\; \prod_{\mathrm{i,j}=1}^{R} \frac{1}{2\cdot b[\mathrm{i,j}]}\, \exp\Big(\, -\big|u[\mathrm{i,j}]-\mu[\mathrm{i,j}]\big| \,/\, b[\mathrm{i,j}] \,\Big) \tag{2.28}$$

$$p_{\text{Cauchy}}(u\,;\,\mu,\gamma) \;=\; \prod_{\mathrm{i,j}=1}^{R} \Bigg(\pi\cdot\gamma[\mathrm{i,j}]\cdot\bigg(1 \,+\, \frac{1}{\gamma[\mathrm{i,j}]^2}\big(u[\mathrm{i,j}] \,-\, \mu[\mathrm{i,j}]\big)^2\bigg)\Bigg)^{-1} \tag{2.29}$$

Experiments with these alternative implementations show that the network is still able to achieve comparable levels of accuracy with respect to the mean squared error. In both cases, however, the predictive uncertainties fail to capture the empirical distribution of the network's errors. Indeed, after inspecting the empirical distribution of the network errors, it is quite evident that the errors remain normally distributed despite the change to the loss function.

In this section, we provide a detailed analysis of these alternative uncertainty models in order to provide insight into how the network's uncertainty schema works. In particular, we show that the network does, in fact, succeed in fitting the respective distributions to the data. However, since the network errors are observed to be normally distributed in practice, the fit of the Laplace and Cauchy distributions do not provide a faithful representation of the network's uncertainty. The true uncertainty can be easily recovered, however, by considering the relationship between the optimal scale parameters of the distributions and the standard deviations of the normal data to which they are fit.

2.B.1 Laplace uncertainty model

To understand how the use of the Laplace loss function affects the uncertainty schema, we consider the result of fitting a Laplace distribution to normally distributed data. Following the approach of Kundu [46], we consider the maximum likelihood estimator (MLE) for the scale parameter b of a mean-zero Laplace distribution corresponding to independent identically distributed observations $\{x_{\mathrm{i}}\}_{\mathrm{i}=1}^{N}$:

$$\mathrm{MLE}(b) \;=\; \frac{1}{N}\sum_{\mathrm{i}=1}^{N} |x_{\mathrm{i}}| \tag{2.30}$$

Converting to a continuous setting, and assuming that the data is distributed as $x \sim \mathcal{N}(0,\sigma)$, we have:

$$\mathrm{MLE}(b) \;=\; \int_{\mathbb{R}} \frac{|x|}{\sqrt{2\pi\sigma^2}}\,\mathrm{e}^{-x^2/2\sigma^2}\,dx \;=\; \sqrt{\frac{2}{\pi}}\int_0^\infty \frac{x}{\sigma}\cdot\mathrm{e}^{-x^2/2\sigma^2}\,dx \;=\; \sqrt{\frac{2}{\pi}}\int_0^\infty \sigma\,\mathrm{e}^{-u}\,du \;=\; \sqrt{\frac{2}{\pi}}\,\sigma \tag{2.31}$$

Thus, the scale parameter b of a Laplace distribution which is inappropriately fit to normally distributed data should converge to the true standard deviation σ scaled by a factor of $\sqrt{2/\pi}$. In particular, we can recover the true standard deviation σ of the normally

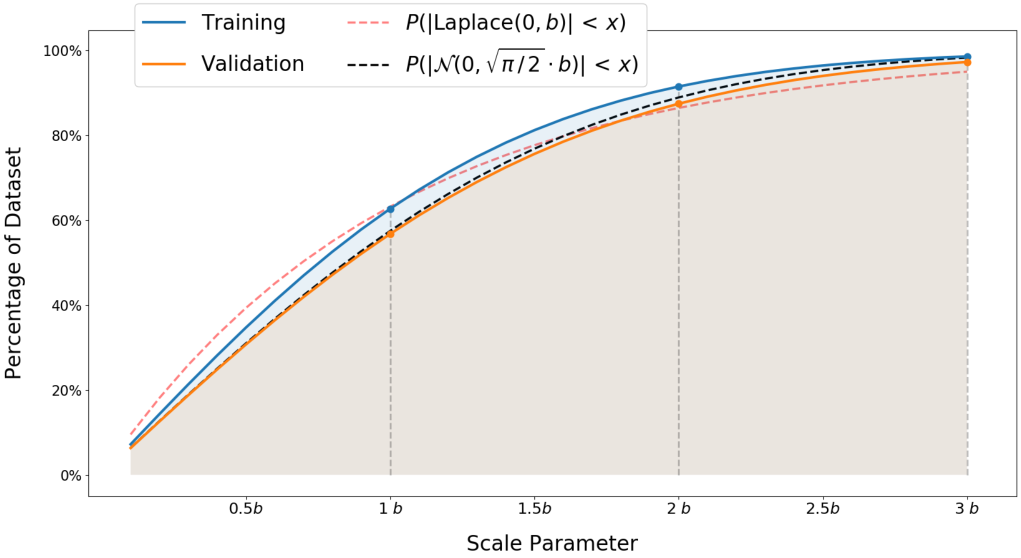


Figure 2.11. Quantitative analysis of the network's predicted uncertainties for the "Varying Domain Poisson Equation" using the Laplace distribution for uncertainty quantification. The network's errors are observed to be better fit by normal distributions with standard deviations rescaled by the factor of $\sqrt{\pi/2}$ derived in the MLE calculation for the Laplace scale parameter b.

distributed data from the MLE calculation for b by simply rescaling by a factor of $\sqrt{\pi/2}$. Indeed, this is precisely the result of training the network in the "Varying Domain Poisson Equation" setup using the Laplace loss function, as illustrated in Figure 2.11.

2.B.2 Cauchy uncertainty model

An analogous analysis can be carried out for the Cauchy uncertainty model by considering the maximum likelihood estimator (MLE) for the scale parameter γ of a mean-zero Cauchy distribution associated with independent identically distributed observations $\{x_\mathrm{i}\}_{\mathrm{i}=1}^{N}$. The MLE criterion for the scale parameter γ has been derived in Equation 2 of [13] and is as follows:

$$\text{MLE}(\gamma) \;=\; \gamma \quad \text{such that} \quad \frac{1}{N}\sum_{\mathrm{i}=1}^{N} \frac{\gamma^2}{x_\mathrm{i}^2 + \gamma^2} \;=\; \frac{1}{2} \tag{2.32}$$

Converting to a continuous setting, and assuming that the data is distributed as $x \sim \mathcal{N}(0, \sigma)$, we have:

$$\text{MLE}(\gamma) \;=\; \gamma \quad \text{such that} \quad \int_{\mathbb{R}} \frac{\gamma^2}{x^2 + \gamma^2} \, \frac{1}{\sqrt{2\pi\sigma^2}} \, e^{-x^2/2\sigma^2} \, dx \;=\; \frac{1}{2} \tag{2.33}$$

Using Equation 7 in Section 3.2 of [65], with $a = 1/\sigma\sqrt{2}$ and $z = \gamma$, the integral in consideration can be expressed in terms of the complementary error function $\text{erfc}(x) = 1 - \text{erf}(x) = 1 - 2/\sqrt{\pi} \int_0^x e^{-t^2} \, dt$:

$$\text{erfc}\left(\frac{\gamma}{\sigma\sqrt{2}}\right) \;=\; \frac{2\gamma}{\pi} \, e^{-\gamma^2/2\sigma^2} \int_0^\infty \frac{e^{-x^2/2\sigma^2}}{x^2 + \gamma^2} \, dx \;=\; \frac{\gamma}{\pi} \, e^{-\gamma^2/2\sigma^2} \int_{\mathbb{R}} \frac{e^{-x^2/2\sigma^2}}{x^2 + \gamma^2} \, dx \tag{2.34}$$

Solving for the integral on the right hand side and rescaling by $\gamma^2/\sqrt{2\pi\sigma^2}$ yields:

$$\int_{\mathbb{R}} \frac{\gamma^2}{x^2 + \gamma^2} \, \frac{1}{\sqrt{2\pi\sigma^2}} \, e^{-x^2/2\sigma^2} \, dx \;=\; \sqrt{\frac{\pi}{2}} \cdot \frac{\gamma}{\sigma} \cdot e^{\gamma^2/2\sigma^2} \cdot \text{erfc}\left(\frac{\gamma}{\sigma\sqrt{2}}\right) \tag{2.35}$$

The optimal value of γ can now be approximated numerically by minimizing the difference between the expression on the right and the target value of 1/2. In particular, we find that $\gamma \approx 0.612$ in the case of a unit standard deviation $\sigma = 1$ and that the scale parameter γ scales linearly with the standard deviation σ.

Thus, the MLE calculation for the scale parameter γ can be approximated by $\gamma \approx 0.612 \cdot \sigma$, and the true standard deviation of the normally distributed data can be recovered via $\sigma \approx 1.634 \cdot \gamma$. This again aligns perfectly with the results of training the network in the "Varying Domain Poisson Equation" setup using the Cauchy loss function, as illustrated in Figure 2.12.

These results strongly suggest that the network has in fact accurately estimated the optimal scale parameters b and γ for the Laplace and Cauchy distributions, respectively. The mismatch between the predicted uncertainties and observed errors is simply a consequence of fitting a non-normal distribution to normally distributed data. As detailed above, the correct uncertainty can be recovered by observing that the errors are in fact normally distributed;

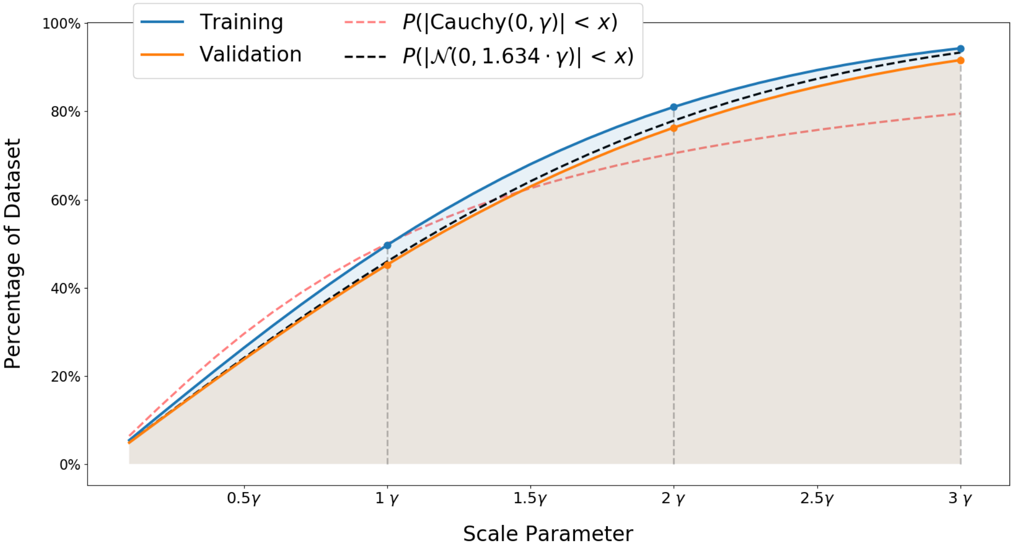


Figure 2.12. Quantitative analysis of the network's predicted uncertainties for the "Varying Domain Poisson Equation" using the Cauchy distribution for uncertainty quantification. The network's errors are seen to be better fit by normal distributions with standard deviations rescaled by the factor of 1.634 derived in the MLE calculation for the Cauchy scale parameter γ.

the correction factors from the MLE calculations can then be used to derive the true standard deviations from the predicted scale parameter values.

3. OPERATOR NETWORKS & PREDICTIVE UNCERTAINTY

> There is certain danger in dealing only with the general theory.
>
> Nicolai Krylov

3.1 Introduction

The DeepONet [60] framework was introduced by Lu et al. in 2019 and has been successfully applied in a number of more recent works [10, 62, 80]. This framework provides a fundamentally different approach for modeling operators which can be seen as a hybrid between conventional approximator networks and the black-box function-to-function mappings considered in the previous chapter[1]. In particular, an independent pair of fully-connected networks are designed to parse function data and domain data separately, and the resulting data is fused together in a precise, interpretable way to produce the network output.

One of the key differences between the DeepONet and ConvPDE-UQ approaches is the format requirements of the input data specifying the PDE systems. While ConvPDE-UQ necessitates input data structured on a uniform grid, DeepONet models are designed to parse observation data retrieved from *sensors* placed throughout the spatial domain. This is particularly useful for many physical applications where sensor data is more commonly available than complete uniform grids of observations.

While the use of fixed sensors is advantageous for many real-world applications, it also restricts the scope of any particular DeepONet model to a specific domain. This is in contrast to the ConvPDE-UQ approach presented in the previous chapter, where the resulting operator networks were designed to be applicable to varied domains after a single offline training procedure. In the context of a fixed domain, however, the performance of DeepONet models is often quite remarkable. In particular, these models are capable of learning

[1] ↑The hybrid interpretation of DeepONet is discussed in more detail in Section 3.4.1

from very limited amounts of training data, a fact that is especially impressive given the unstructured nature of the input data provided to the networks.

In this chapter, we investigate the possibility of equipping DeepONet models with predictive uncertainty capabilities analogous to those provided by ConvPDE-UQ networks. A particular emphasis is placed on PDE systems with inhomogeneous boundary conditions, and a modified data generation procedure is proposed to remove the dependence on conventional numerical methods for training DeepONet models. We also present computational techniques for optimizing the resulting operator networks to accommodate real-time inference tailored to specific applications and use-cases.

3.1.1 DeepONet framework

DeepONet architectures are comprised of two central components referred to as the *branch network* and *trunk network*. The branch network is responsible for processing information associated with input functions, and the trunk network parses input coordinates corresponding to evaluation locations on the spatial domain. The branch and trunk components produce vectors b and t which summarize the information/features extracted by the respective networks; these features are then combined to form the final network prediction, which is defined by the inner product $\widehat{u} = \langle\, b \,,\, t \,\rangle$.

While this predefined method of combining the outputs of the neural network components slightly restricts the expressivity of the resulting model, it also provides a distinctive advantage with respect to interpretability. Since the output t of the trunk network depends solely on the input coordinates x, it is natural to interpret the components of the trunk output as functions defined on the underlying spatial domain of the PDE.

$$\text{Trunk}(x) \;\;=\;\; \Big[\;— \;t\; —\; \Big] \;\;=\;\; \Big[\varphi_1(x) \,,\, \ldots \,,\, \varphi_N(x) \Big] \tag{3.1}$$

As illustrated in Figure 3.1, the component functions of the trunk network often bear a strong resemblance to the basis functions from the finite element method reviewed in Section 1.3.3. In the context of DeepONet models, however, these basis functions are learned by the trunk network automatically during training, as opposed to being specified beforehand as is the case in FEM calculations. The branch network then assumes the role of computing the appropriate coefficients b for the basis functions produced by the trunk, ensuring that the resulting network approximation:

$$\widehat{u}(x) \;=\; \langle\, b\,,\, t\,\rangle \;=\; \textstyle\sum_{\mathrm{i}=1}^{N} b_{\mathrm{i}} \cdot \varphi_{\mathrm{i}}(x) \tag{3.2}$$

satisfies the PDE constraints specified by the input functions under consideration.

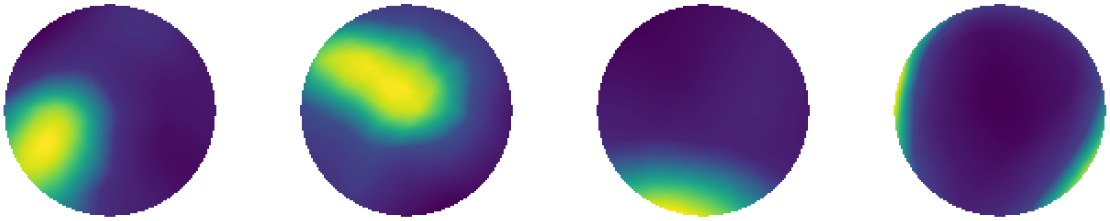


Figure 3.1. Example basis functions learned by the trunk network of a DeepONet model.

3.1.2 PDE systems and data generation

In this chapter, we present the results of experiments conducted using DeepONet models to approximate the solution operators associated with the following problem setups:

Poisson (Homogeneous BC)

$$\begin{cases} \Delta u \,=\, f & \text{in} \quad \Omega \\ \quad u \,=\, 0 & \text{on} \quad \partial\Omega \end{cases}$$

Poisson (Inhomogeneous BC)

$$\begin{cases} \Delta u \,=\, f & \text{in} \quad \Omega \\ \quad u \,=\, g & \text{on} \quad \partial\Omega \end{cases}$$

Nonlinear Poisson

$$\begin{cases} \operatorname{div}\Big(\big(1+u^2\big)\,\nabla u\Big) \,=\, f & \text{in} \quad \Omega \\ \qquad\qquad\qquad\quad u \,=\, g & \text{on} \quad \partial\Omega \end{cases}$$

Nonlinear Diffusion-Reaction

$$\begin{cases} \lambda \cdot \Delta u \,+\, u^2 \,=\, f & \text{in} \quad \Omega \\ \qquad\qquad\quad u \,=\, g & \text{on} \quad \partial\Omega \end{cases}$$

The experiments conducted for the purposes of this work will focus on problems defined on the unit square and unit circle in this work; however, the framework and network implementations are equally applicable to general connected and bounded domains.

As noted in the previous chapter, the incorporation of inhomogeneous boundary conditions presents a key technical challenge in the context of neural network approximations for PDEs. In particular, handling input data associated with inhomogeneous boundary conditions is complicated by the fact that the interior and boundary data come from two distinct functions, often with different scales/properties. In addition, the dimensionality of boundary data is strictly less than that of the interior data, as shown in Figure 3.2. This leads to a sparse representation of the boundary data in the ambient space which poses a significant challenge for convolutional architectures.

In the context of fully-connected layers, however, the boundary can simply be appended to the list of sensor values associated with the interior data. While there are a number of natural architecture choices for handling interior and boundary data, as described in Appendix 3.A.1, experimental results show that simply appending the boundary data to the network input vector achieves the same level of performance as more structured approaches.

To generate the training datasets for the DeepONet models, we make use of the Gaussian process sampling procedure introduced in Section 2.3.5 of the previous chapter. Instead of using these samples as realizations of PDE coefficients, however, we will employ the samples as realizations of the solution functions. In particular, we sample:

$$u \;\sim\; \mathcal{G}(0, k_l(x,y)) \tag{3.3}$$

from mean-zero Gaussian random fields with squared exponential covariance kernels $k_l(x,y) = \exp(-\|x-y\|^2/2l^2)$ and length-scales $l \in \{0.2, 0.2333, 0.2667, 0.3\}$. The associated interior data f and boundary data g can then be computed by applying the forward differential operator and boundary operator directly to the sampled solution. Importantly, this variation on the data generation procedure allows us to construct training datasets without relying on conventional numerical solvers, such as FEM implementations; the datasets can be constructed whenever numerical implementations of the forward differential operator and boundary operator are available. In addition, with regard to inhomogeneous boundary conditions, this procedure ensures that the interior data f and boundary data g correspond to a well-defined solution u. This is of particular importance for more general nonlinear equations where non-trivial relationships between the interior and boundary data may be required for a solution to exist.

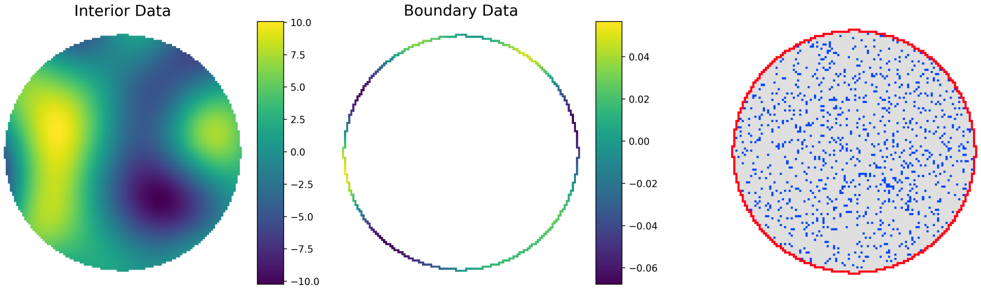


Figure 3.2. Example input data consisting of 2-dimensional interior data and 1-dimensional boundary data; DeepONet sensor locations for the interior data (blue) and boundary data (red) are shown on the right.

3.2 Architecture for predictive uncertainties

In this section, we show that the DeepONet architecture can be extended to provide predictive uncertainties following a procedure analogous to that used in the ConvPDE-UQ framework. After conducting experiments with several architecture variations, as described in Appendix 3.A, the network structure outlined in Figure 3.3 was found to provide the best performance.

The branch and trunk networks each consist of 5 fully-connected layers with 50 units and 30 units per layer, respectively. Each network layer uses Leaky ReLU activation functions, with the exception of the final layer of the branch network where no activation is applied. In order to equip the networks with predictive uncertainties, the final two layers of the branch and trunk are split to provide independent processing for the mean and uncertainty parameters. More specifically, two distinct pairs of 50 unit layers are used to produce the mean output b and uncertainty output b_σ of the branch network; similarly, two distinct pairs

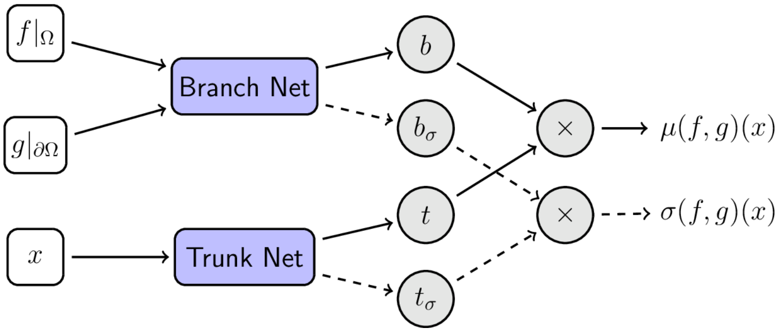


Figure 3.3. Probabilistic network architecture for DeepONet models with predictive uncertainty.

of 30 unit layers are used to produce the trunk outputs t and t_σ. For the experimental results presented in this work, the number of basis functions N was taken to be 150; however, it should be noted that the performance of the models remain essentially unchanged for any choice of $N > 50$.

3.2.1 Loss functions and training procedure

While the network architecture proposed in the previous section allows the DeepONet networks to attach uncertainty parameters to the model predictions, it still remains to be shown how these parameters can be calibrated during training to produce accurate uncertainty estimates. In order translate the uncertainty parameters into uncertainty estimates, it suffices to select an appropriate parameterized family of probability distributions.

In practice, this can be done by observing the empirical distribution of network errors observed during training. More precisely, a histogram of the observed network errors during training, such as the one shown in Figure 3.4, can be compared with the density functions of common families of distributions, such as those presented in Figure 1.8. Once an appropriate family of parameterized probability distributions is selected, the network loss is defined with respect to the associated log likelihood function, following the analogy of Gaussian process regression used in the ConvPDE-UQ framework. In particular, when the network errors are observed to follow a normal distribution, the loss function is given explicitly by:

$$\text{Loss} \;=\; \tfrac{1}{2}\Big(\mu(f,g)(x) \,-\, u(f,g)(x)\Big)^2 \Big/ \Big(\sigma(f,g)(x)\Big)^2 \;+\; \log\Big(\sigma(f,g)(x)\Big) \;+\; \tfrac{1}{2}\log(2\pi) \quad (3.4)$$

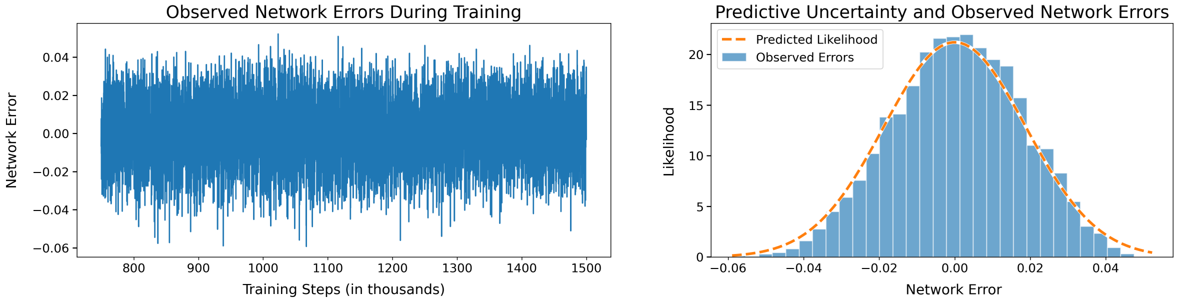


Figure 3.4. Observed network errors during training and final uncertainty predictions.

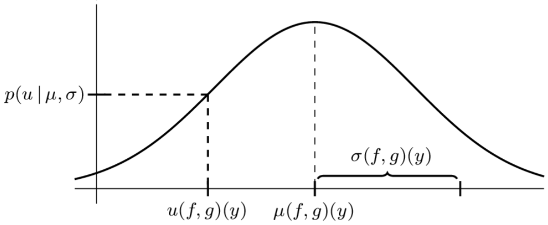


Figure 3.5. Interpretation of network outputs as parameters for a normal distribution.

The network outputs $\mu(f,g)(x)$ and $\sigma(f,g)(x)$ are thus interpreted as parameters for a predictive probability distribution placed on the unknown values of the true solution. By construction, the loss function corresponds to the negative log likelihood, $-\log p(u\,|\,\mu,\sigma)$, associated with an observation from the true solution, $u(f,g)(x)$, as illustrated in Figure 3.5.

3.2.2 Predictive uncertainty and function space generalization

While the proposed network architecture and training procedure allow DeepONet models to make uncertainty estimates, verifying that these estimates generalize beyond the scope of the data provided during training is of fundamental importance. In the context of the proposed data generation and training procedure, it is natural to consider two distinct cases of generalization: (1) performance on unseen validation data sampled from the length-scale classes used during training and (2) performance on testing data sampled outside of the length-scale range seen by the network. The former case corresponds to the network generalization on in-distribution data, while the latter corresponds to the generalization for out-of-distribution (OoD) data [25].

In the presence of sufficient training data, the prediction accuracy of the DeepONet models on in-distribution validation data matches the accuracy on training examples across all length-scales. The associated uncertainty estimates are also well-calibrated to the validation examples provided the availability of sufficient data during training, as shown in Figure 3.6.

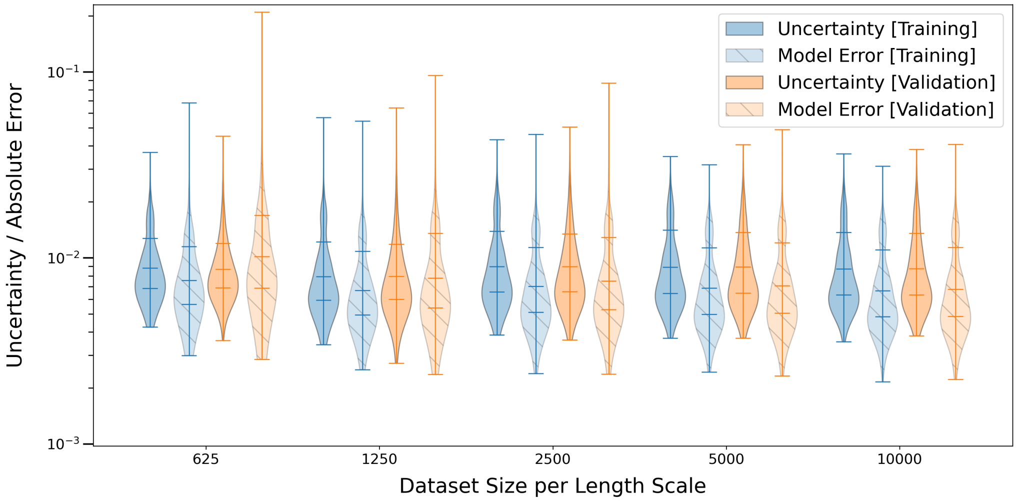


Figure 3.6. Comparison of DeepONet performance on training and validation datasets.

Of greater interest is the fact that the predictive uncertainties generalize quite well to out-of-distribution data, as shown in Figure 3.7. As noted in the previous chapter, the lower length-scale classes correspond to more difficult problems; accordingly, the network performance gradually decays as the length-scale is reduced. More importantly, the predictive uncertainties for the smallest length-scales are also observed to increase in agreement with the rising network errors associated with the out-of-distribution data. However, this rise in uncertainty does not occur for the out-of-distribution at the other extreme (i.e. length-scales larger than those seen during training), which correctly reflects the model performance for these simpler problems.

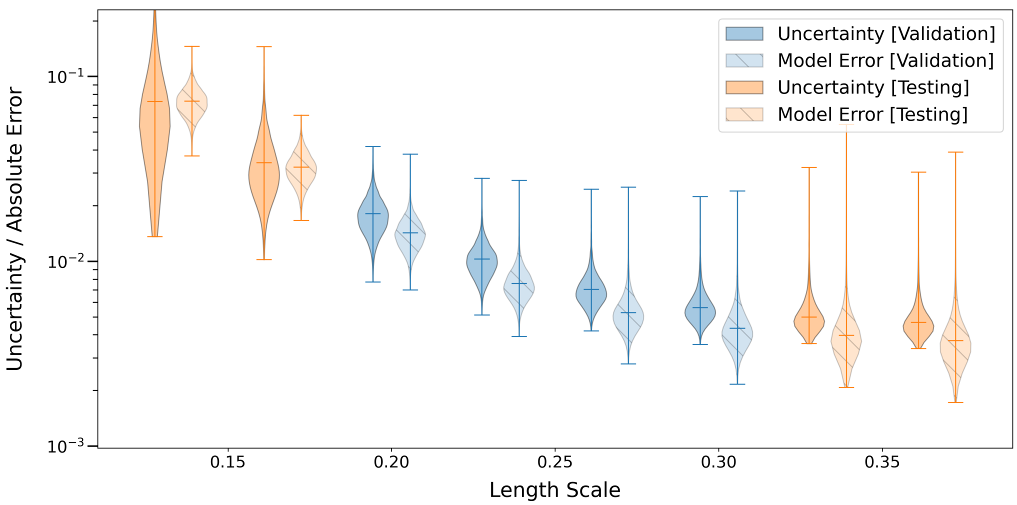


Figure 3.7. Analysis of predictive uncertainty produced by a DeepONet model on the Nonlinear Diffusion-Reaction problem setup.

From this, we see that the network is, to some extent, able to automatically determine the difficulty associated with a given problem. More precisely, the accuracy of network's predictive uncertainties extend beyond the scope of the function classes observed during training. Notably, the network is never provided any labeling information regarding the length-scale values, so the network's predictive uncertainty is based solely on information/features extracted from the input function values.

3.3 Summary of numerical results

The numerical results for the experiments conducted using DeepONet models to approximate solutions to the PDE systems listed in Section 3.1.2 are summarized in the tables below. Each table represents a specific problem setup, and the network performance is provided for each length-scale class. As noted in the previous section, the length-scales $\{0.2000, 0.2333, 0.2667, 0.3000\}$ correspond to the function spaces observed during training while the length-scales $\{0.1333, 0.1667, 0.3333, 0.3667\}$ correspond to out-of-distribution data. We note that in general, the DeepONet models are capable of achieving close to a 1% relative error for all problems, with performance worsening as the length-scale is reduced. Qualitative results for the Nonlinear Poisson problem setup are also provided in Figure 3.8.

Table 3.1. Square with Dirichlet boundary conditions.

	0.1333	0.1667	0.2000	0.2333	0.2667	0.3000	0.3333	0.3337
MSE	4.78e-04	1.45e-04	5.17e-05	2.82e-05	2.10e-05	1.78e-05	1.60e-05	1.50e-05
MAE	1.71e-02	9.38e-03	5.60e-03	4.11e-03	3.51e-03	3.20e-03	3.01e-03	2.89e-03
L^1 Rel	3.33e-02	1.61e-02	8.60e-03	5.82e-03	4.62e-03	3.96e-03	3.63e-03	3.31e-03
L^2 Rel	3.19e-02	1.56e-02	8.36e-03	5.69e-03	4.55e-03	3.93e-03	3.62e-03	3.32e-03

Table 3.2. Square with inhomogeneous boundary conditions.

	0.1333	0.1667	0.2000	0.2333	0.2667	0.3000	0.3333	0.3337
MSE	1.94e-02	4.17e-03	7.90e-04	1.67e-04	5.46e-05	3.22e-05	9.35e-05	8.83e-05
MAE	8.78e-02	3.80e-02	1.59e-02	7.21e-03	4.09e-03	2.95e-03	2.91e-03	2.63e-03
L^1 Rel	2.25e-01	9.87e-02	4.19e-02	1.92e-02	1.10e-02	8.04e-03	7.67e-03	6.91e-03
L^2 Rel	2.83e-01	1.32e-01	5.88e-02	2.76e-02	1.59e-02	1.15e-02	1.07e-02	9.66e-03

Table 3.3. Circle with inhomogeneous boundary conditions.

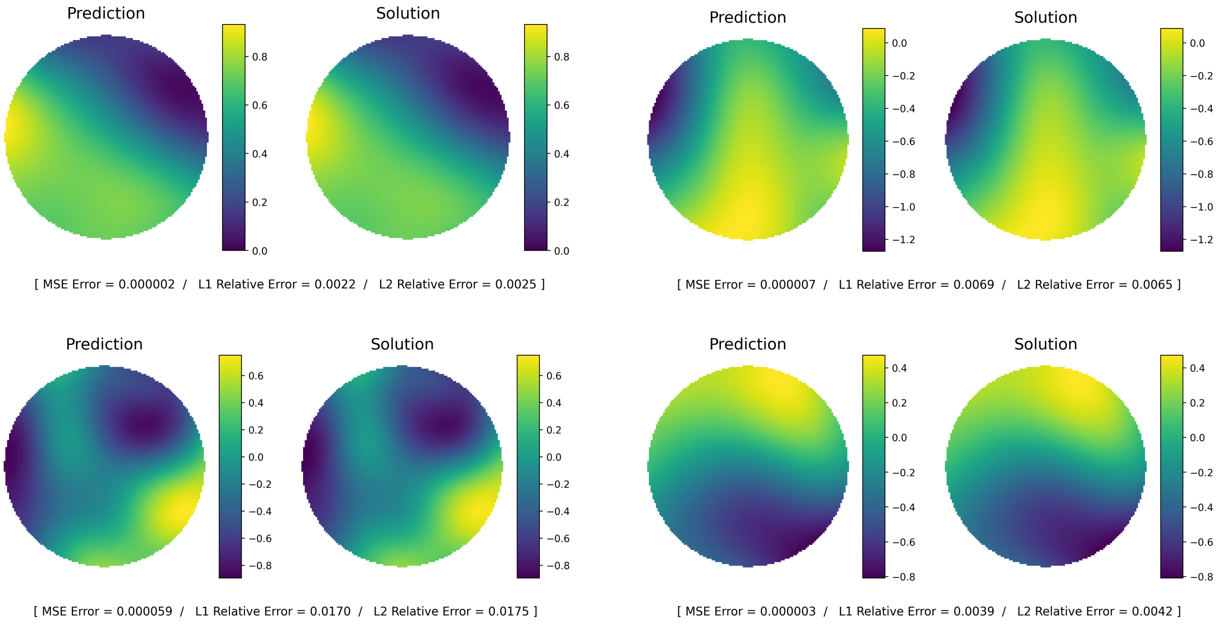
	0.1333	0.1667	0.2000	0.2333	0.2667	0.3000	0.3333	0.3337
MSE	3.85e-03	5.23e-04	7.04e-05	1.43e-05	6.45e-06	3.08e-06	3.69e-06	3.66e-06
MAE	4.63e-02	1.66e-02	6.07e-03	2.70e-03	1.63e-03	1.22e-03	1.07e-03	9.37e-04
L^1 Rel	1.20e-01	4.40e-02	1.64e-02	7.43e-03	4.51e-03	3.42e-03	3.06e-03	2.60e-03
L^2 Rel	1.28e-01	4.79e-02	1.83e-02	8.30e-03	4.97e-03	3.75e-03	3.35e-03	2.86e-03

Table 3.4. Nonlinear with inhomogeneous boundary conditions.

	0.1333	0.1667	0.2000	0.2333	0.2667	0.3000	0.3333	0.3337
MSE	4.46e-03	4.79e-04	6.71e-05	1.64e-05	9.30e-06	5.92e-06	1.97e-05	1.70e-05
MAE	4.95e-02	1.67e-02	6.24e-03	3.01e-03	2.00e-03	1.58e-03	1.54e-03	1.41e-03
L^1 Rel	1.29e-01	4.43e-02	1.67e-02	8.07e-03	5.31e-03	4.21e-03	3.94e-03	3.66e-03
L^2 Rel	1.32e-01	4.61e-02	1.77e-02	8.58e-03	5.64e-03	4.48e-03	4.20e-03	3.89e-03

Table 3.5. Nonlinear Diffusion-Reaction with inhomogeneous boundary conditions.

	0.1333	0.1667	0.2000	0.2333	0.2667	0.3000	0.3333	0.3337
MSE	6.87e-03	1.28e-03	2.70e-04	8.83e-05	6.65e-05	5.77e-05	4.87e-05	5.67e-05
MAE	6.50e-02	2.77e-02	1.22e-02	6.48e-03	4.57e-03	3.89e-03	3.61e-03	3.51e-03
L^1 Rel	1.69e-01	7.27e-02	3.24e-02	1.69e-02	1.17e-02	9.85e-03	9.26e-03	8.97e-03
L^2 Rel	1.71e-01	7.47e-02	3.37e-02	1.77e-02	1.23e-02	1.04e-02	9.84e-03	9.55e-03



[MSE Error = 0.000002 / L1 Relative Error = 0.0022 / L2 Relative Error = 0.0025]

[MSE Error = 0.000007 / L1 Relative Error = 0.0069 / L2 Relative Error = 0.0065]

[MSE Error = 0.000059 / L1 Relative Error = 0.0170 / L2 Relative Error = 0.0175]

[MSE Error = 0.000003 / L1 Relative Error = 0.0039 / L2 Relative Error = 0.0042]

Figure 3.8. DeepONet results on validation dataset for Nonlinear equation on unit circle.

3.4 Optimization for inference

When deploying DeepONet models for real-time applications, it is often important to modify the computational workflow based off of the target application. In this section, we consider to common use-cases where redundant calculations can be avoided in order to achieve significantly faster inference speeds. We first consider the construction of light-weight surrogate models for specific PDE solutions under the assumption of fixed input data. This is followed by an overview of the steps required for performing fast inference on the full spatial domain for arbitrary input data.

3.4.1 Precomputing branch weights

The first use-case we will consider involves the construction of light-weight surrogate networks which can be used to approximate the solution function associated with fixed input data (f, g). The optimization required for this type of application is quite trivial, but yields a noticeable improvement in inference speed and helps motivate the more involved optimization steps introduced in the following section.

In this setting, we note that the branch output b remains constant for any choice of evaluation location x (since the branch depends only on the fixed input data (f,g)). By precomputing the branch weights, the solution $u(f,g)$ can be evaluated at arbitrary evaluation locations x using just a single forward-pass of the trunk network, as shown in Figure 3.9. In this context, the DeepONet framework can be seen as a natural generalization of the conventional "approximator" networks described in Section 1.4.2; i.e. once the branch weights are fixed, the DeepONet framework yields fast neural network surrogate models[2].

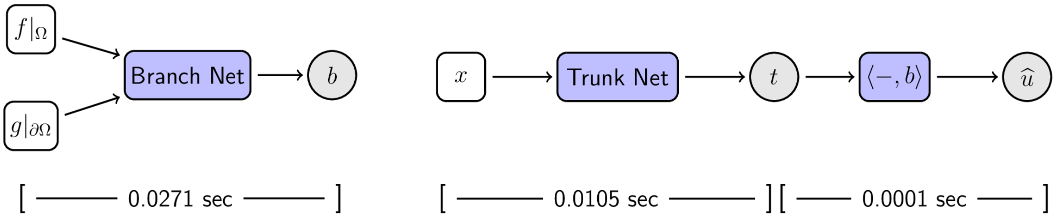


Figure 3.9. Inference speeds for DeepONet surrogates with precomputed branch outputs. The resulting surrogate model provides an efficient method for evaluating the approximate solution at arbitrary spatial locations.

Moreover, these light-weight surrogates provide access to potentially valuable derivative information with very minimal overhead. In particular, since the branch output b is independent of the input coordinate x and the basis functions $\varphi_\mathrm{i}(x)$ are defined by the neural network architecture of the trunk component, a linear differential operator $\mathcal{L}$ can be applied to DeepONet surrogates by computing:

$$\mathcal{L}_x[u(f,g)(x)] \;=\; \mathcal{L}_x\left[\textstyle\sum_{\mathrm{i}=1}^{N} b_\mathrm{i}\cdot\varphi_\mathrm{i}(x)\right] \;=\; \textstyle\sum_{\mathrm{i}=1}^{N} b_\mathrm{i}\cdot\mathcal{L}_x\left[\varphi_\mathrm{i}(x)\right] \tag{3.5}$$

where $\mathcal{L}_x\left[\varphi_\mathrm{i}(x)\right]$ is evaluated using automatic differentiation through the trunk network.

3.4.2 Precomputing trunk weights

Another common use-case for DeepONet models concerns scenarios where the input data (f,g) is expected to vary, and we are interested in evaluating the associated solutions

[2] ↑However, the key difference is that "approximator" networks need to be re-trained whenever the input data (f,g) is modified, while DeepONet surrogates can be obtained by simply recomputing the branch weights.

quickly at a fixed set of locations throughout the spatial domain. As one may expect, the optimization relevant for this form of inference is primarily based on precomputing the trunk weights. In order to achieve real-time speeds in practice, however, it also necessary to make a few additional adjustments to inference calculations.

To begin, we review the computational steps involved with evaluating a DeepONet model with input data (f, g) at a fixed set of evaluation locations $\{x_\mathrm{j}\}_{\mathrm{j}\in\mathcal{E}}$. First, we need to evaluate the trunk at each of the input locations to form the matrix:

$$T \;=\; \begin{bmatrix} — & t_1 & — \\ \vdots & \vdots & \vdots \\ — & t_E & — \end{bmatrix} \qquad \text{where} \qquad t_\mathrm{j} \;=\; \Big[t_{\mathrm{j}1}\,,\;\ldots\;,\; t_{\mathrm{j}N} \Big] \;=\; \mathrm{Trunk}(x_\mathrm{j}) \tag{3.6}$$

The branch output vector is then computed for the current input data (f, g), and the network prediction $\widehat{u}$ is given in vector form by:

$$\begin{bmatrix} \widehat{u}_1 \\ \vdots \\ \widehat{u}_E \end{bmatrix} \;=\; \begin{bmatrix} — & t_1 & — \\ \vdots & \vdots & \vdots \\ — & t_E & — \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix} \quad \text{where} \quad b^T \;=\; [\, b_1\,,\;\ldots\;,\; b_N\,] \;=\; \mathrm{Branch}(f, g) \tag{3.7}$$

In order to interpret the vector prediction $\widehat{u}$ as a function on a two-dimensional domain, we must then place each of the entries $\widehat{u}(x_\mathrm{j})$ in a data structure which specifies the associated spatial position x_j. More concretely, it is often desirable to have an array representation of the predicted solution $\widehat{u}$; in this case, the evaluation locations $\{x_\mathrm{j}\}_{\mathrm{j}\in\mathcal{E}}$ can be defined with respect to a uniform grid, with points outside of the domain omitted. To efficiently convert the vector prediction into a structured array, it is advisable to construct a sparse “placement matrix” P defined by mapping each index $\mathrm{j} \in \mathcal{E}$ to the appropriate array location. The full computational workflow for this form of inference is summarized in Figure 3.10.

The computation time resulting from the naïve implementation above is not very well-suited for real-time applications; however, the inference time can be greatly reduced when the evaluation locations are fixed. In particular, after the DeepONet training procedure is completed the trunk network outputs $\{t_\mathrm{j}\}_{\mathrm{j}\in\mathcal{E}}$ and placement matrix P need only be computed once to facilitate inference for arbitrary input data (f, g). As shown in Figure 3.11, the computation time required to evaluate the DeepONet approximation across the full domain can be reduced to approximately 0.03 seconds per problem.

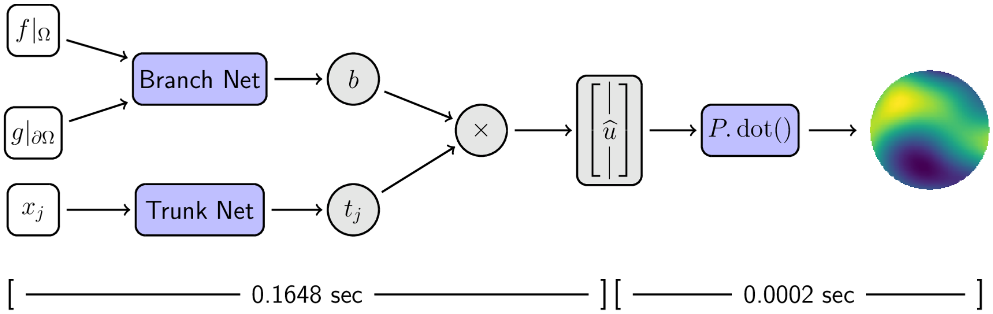


Figure 3.10. Standard inference speeds without precomputing trunk outputs; the placement matrix P is used to efficiently restructure the vectorized network outputs to match the format of the target two-dimensional domain.

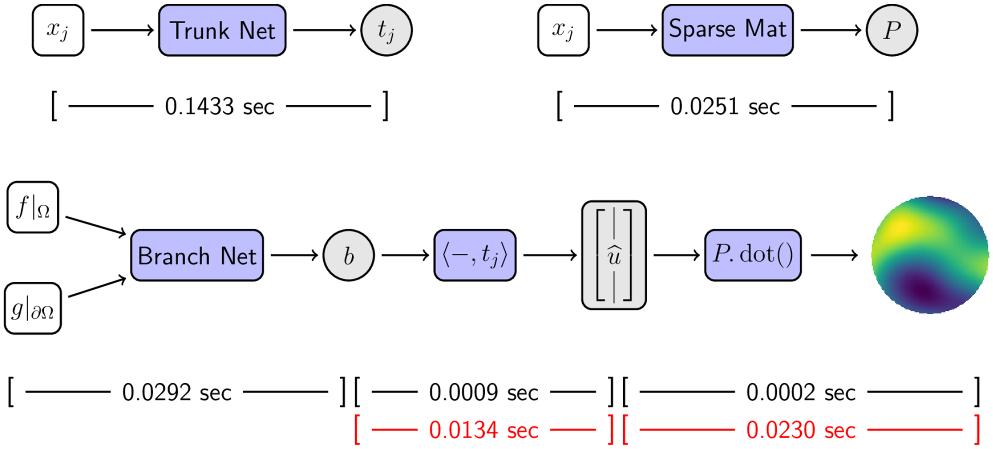


Figure 3.11. Inference can be optimized for fast evaluations on a fixed grid by precomputing the trunk outputs associated with each grid location and constructing a sparse placement matrix for mapping vectorized network outputs to the correct array locations. The timings in the top row of the figure are computed only once after training. The timings shown in red denote the unoptimized speeds using loops and manual entry placement instead of einsum operations and sparse matrix-vector products with the placement matrix.

3.4.3 Concluding remarks

In this chapter, we have shown that the DeepONet framework for the construction of operator networks can be extended to produce predictive uncertainty estimates with minimal modifications to the underlying network architecture. The resulting uncertainty estimates were shown to be well-calibrated to the observed network error for the function classes seen by the network during training, and remained well-calibrated when tested on out-of-distribution data as well. These fully-connected operator networks were also shown to provide accurate approximations to PDE systems involving inhomogeneous boundary conditions. Notably, these networks are capable of working with unstructured sensor data, in contrast to the models introduced in the previous chapter which required uniform observation data. We also demonstrated how the inference speeds associated with trained DeepONet models can be significantly reduced by a careful consideration of the computational workflow used to evaluate the models. Overall, the DeepONet models presented in this chapter were observed to produce approximations with accuracies on the order of 1% relative error and computation times of roughly 0.03 seconds per prediction. Based on these observations, combined with the ability of DeepONet models to work with unstructured sensor data, the DeepONet framework provides a very promising tool for constructing light-weight models which are ideally suited for real-time applications.

3.A Architecture Variations

3.A.1 Incorporation of boundary conditions

A series of ablation studies were conducted based on the following three network architectures regarding the incorporation of inhomogeneous boundary conditions. The architecture illustrated in Figure 3.12 was ultimately chosen due to its simplicity; while the other two variations achieved similar performance, there was no observable benefit associated with separate processing components used for the interior data $f(x)$ and boundary data $g(x)$.

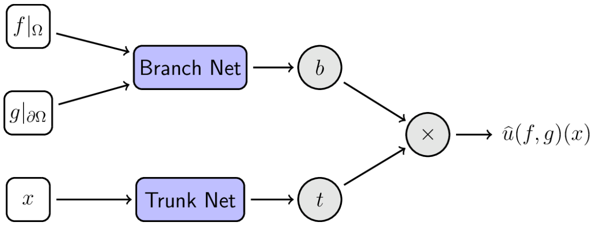


Figure 3.12. DeepONet BC architecture: Variation 1.

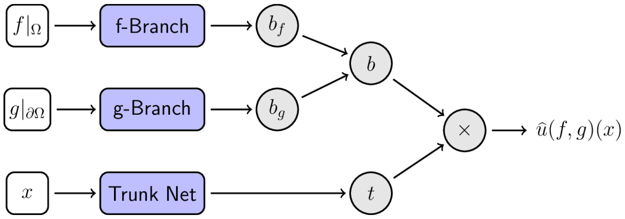


Figure 3.13. DeepONet BC architecture: Variation 2.

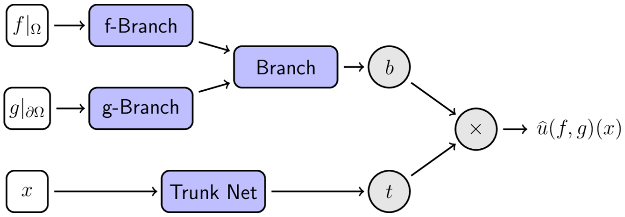


Figure 3.14. DeepONet BC architecture: Variation 3.

3.A.2 Network structures for predictive uncertainties

Once the architecture for the inclusion of boundary conditions was selected, additional ablation experiments were conducted in order to identify the best network design regarding uncertainty predictions. In particular, the three architecture variations illustrated below were considered and the network structure shown in Figure 3.16 was observed to produce the best performance.

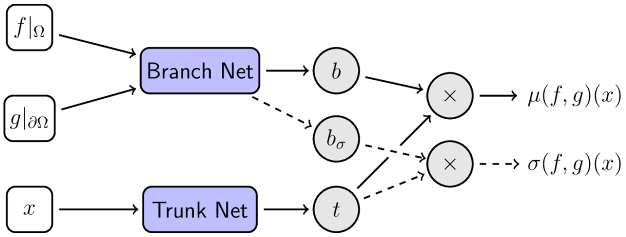


Figure 3.15. DeepONet UQ architecture: Variation 1.

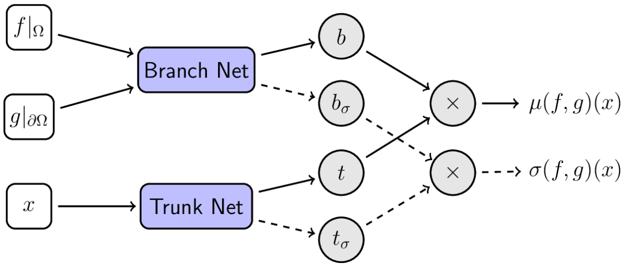


Figure 3.16. DeepONet UQ architecture: Variation 2.

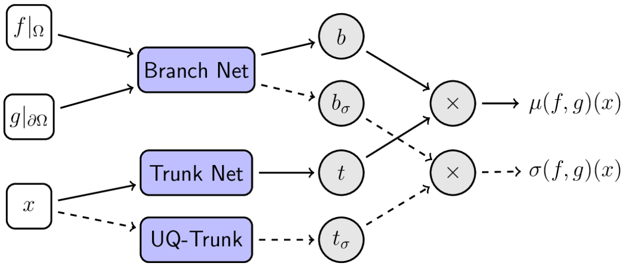


Figure 3.17. DeepONet UQ architecture: Variation 3.

4. SUMMARY AND CONCLUDING REMARKS

That coursing on, whate'er men's speculations,
Amid the changing schools, theologies, philosophies,
Amid the bawling presentations new and old,
The round earth's silent vital laws, facts, modes continue.

Walt Whitman

4.1 Research summary

In this work, we have introduced the ConvPDE-UQ framework for constructing operator networks designed to produce rapid approximations to the solutions of PDE systems. Inspired by the concept of Gaussian process regression, we introduced a training procedure and network architecture designed to produce predictive uncertainty estimates calibrated to the observed model error. This predictive uncertainty schema was shown to produce accurate estimates in the context of the convolutional ConvPDE-UQ framework, and was also shown to be effective in the context of the fully-connected DeepONet framework. We have also demonstrated how ConvPDE-UQ models can be applied to problems on varied domains using a single offline training procedure. Moreover, it was established that DeepONet models can be used to approximate solutions to PDE systems with inhomogeneous boundary conditions based off of unstructured sensor data. The inference speeds of both frameworks were also assessed, and with proper implementations it was shown that both approaches can perform inference in just a few hundredths of a second.

4.2 Future work and applications

The rapid inference speeds of the operator networks considered in this work suggest that these networks may be well-suited for real-time applications; more specifically, in situations where strict time-constraints prevent the use of conventional numerical methods for PDEs it is envisioned that operator network surrogates can be employed as light-weight alternatives. Current research efforts are also being directed toward the inclusion of operator networks for applications involving digital twin models, where we aim to leverage neural network surrogates for processing real-time observation data. In addition, future work will be directed toward applications for more involved optimization procedures where the solutions to PDE systems are required as intermediate steps as a part of a larger calculation and must be approximated quickly in order for optimization to converge within a reasonable time frame.

4.3 Further considerations and limitations

While the proposed operator networks were observed to produced accurate predictions for all of the experiments and problem setups conducted in this work, it is expected that these frameworks will be subject to some practical limitations due to the data-driven nature of model calibration. For example, training networks to enforce constraints of the form:

$$a_{11} \cdot \tfrac{\partial^2 u}{\partial x_1^2} \;+\; a_{12} \cdot \tfrac{\partial^2 u}{\partial x_1 \partial x_2} \;+\; a_{22} \cdot \tfrac{\partial^2 u}{\partial x_2^2} \;+\; a_1 \cdot \tfrac{\partial u}{\partial x_1} \;+\; a_2 \cdot \tfrac{\partial u}{\partial x_2} \;+\; a_0 \cdot u \;\;=\;\; f \tag{4.1}$$

may require a substantial increase in the dataset size in order to capture the relationships between all of the input sources. The randomization procedures used to generate these input sources may also require additional attention in order to ensure that the resulting operator network is properly calibrated to the types of input data that will be encountered in practice.

The data-driven framework used to train operator networks also offers some less obvious advantages in terms of model flexibility. For example, when modeling a constraint of the form $\operatorname{div}(a \cdot \operatorname{grad} u) = f$, the same dataset used to model the forward mapping $(a, f) \mapsto u$ can hypothetically be used to approximate the inverse mapping $(u, f) \mapsto a$. In fact, simply transposing the terms 'a' and 'u' in the training dataset can produce accurate network approximations to this type of inverse problem based on preliminary experiments. Due to the lack of a proper theoretical justification, and related issues such as non-uniqueness, these applications were not considered in the present work; however, initial experiments suggest there is some potential for pursuing this direction, and a more careful analysis in future works may reveal additional applications for operator networks in the context of inverse problems.

REFERENCES

[1] Martin S. Alnæs. “UFL: a Finite Element Form Language”. In: *Automated Solution of Differential Equations by the Finite Element Method, Volume 84 of Lecture Notes in Computational Science and Engineering*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Springer, 2012. Chap. 17.

[2] Martin S. Alnæs, Anders Logg, and Kent-Andre Mardal. “UFC: a Finite Element Code Generation Interface”. In: *Automated Solution of Differential Equations by the Finite Element Method, Volume 84 of Lecture Notes in Computational Science and Engineering*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Springer, 2012. Chap. 16.

[3] Martin S. Alnæs and Kent-Andre Mardal. “On the Efficiency of Symbolic Computations Combined With Code Generation for Finite Element Methods”. In: *ACM Transactions on Mathematical Software* 37.1 (2010). DOI: 10.1145/1644001.1644007.

[4] Martin S. Alnæs and Kent-Andre Mardal. “SyFi and SFC: Symbolic Finite Elements and Form Compilation”. In: *Automated Solution of Differential Equations by the Finite Element Method, Volume 84 of Lecture Notes in Computational Science and Engineering*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Springer, 2012. Chap. 15.

[5] Martin S. Alnæs et al. “The FEniCS Project Version 1.5”. In: *Archive of Numerical Software* 3.100 (2015). DOI: 10.11588/ans.2015.100.20553.

[6] Martin S. Alnæs et al. “Unified Form Language: A domain-specific language for weak formulations of partial differential equations”. In: *ACM Transactions on Mathematical Software* 40.2 (2014). DOI: 10.1145/2566630.

[7] Martin S. Alnæs et al. “Unified Framework for Finite Element Assembly”. In: *International Journal of Computational Science and Engineering* 4.4 (2009), pp. 231–244. DOI: 10.1504/IJCSE.2009.029160.

[8] Robert B Ash et al. *Probability and measure theory*. Academic Press, 2000.

[9] Atilim Gunes Baydin et al. “Automatic differentiation in machine learning: a survey”. In: *Journal of machine learning research* 18 (2018).

[10] Shengze Cai et al. “DeepM&Mnet: Inferring the electroconvection multiphysics fields based on operator approximation by neural networks”. In: *Journal of Computational Physics* 436 (2021), p. 110296.

[11] Tianqi Chen, Haichen Shen, and Arvind Krishnamurthy. *CSE599W Lecture 4: Backpropagation and Automatic Differentiation*. Spring 2018.

[12] Oksana A Chkrebtii et al. "Bayesian solution uncertainty quantification for differential equations". In: *Bayesian Analysis* 11.4 (2016), pp. 1239–1267.

[13] JB Copas. "On the unimodality of the likelihood for the Cauchy distribution". In: *Biometrika* 62.3 (1975), pp. 701–704.

[14] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[15] MWMG Dissanayake and N Phan-Thien. "Neural-network-based approximations for solving partial differential equations". In: *communications in Numerical Methods in Engineering* 10.3 (1994), pp. 195–201.

[16] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of machine learning research* 12.7 (2011).

[17] L.C. Evans and American Mathematical Society. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 1998. ISBN: 9780821807729. URL: https://books.google.com/books?id=5Pv4LVB%5C_m8AC.

[18] Gregory E Fasshauer and Michael J McCourt. *Kernel-based approximation methods using Matlab*. Vol. 19. World Scientific Publishing Company, 2015.

[19] Ken-Ichi Funahashi. "On the approximate realization of continuous mappings by neural networks". In: *Neural networks* 2.3 (1989), pp. 183–192.

[20] Yarin Gal and Zoubin Ghahramani. "Dropout as a Bayesian approximation: Representing model uncertainty in deep learning". In: *international conference on machine learning*. 2016, pp. 1050–1059.

[21] A Ronald Gallant and Halbert White. "There exists a neural network that does not make avoidable mistakes." In: *ICNN*. 1988, pp. 657–664.

[22] David Gilbarg and Neil S Trudinger. *Elliptic partial differential equations of second order*. springer, 2015.

[23] Gene H Golub and Charles F Van Loan. *Matrix computations. Johns Hopkins studies in the mathematical sciences*. 1996.

[24] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[25] Dan Hendrycks and Kevin Gimpel. “A baseline for detecting misclassified and out-of-distribution examples in neural networks”. In: *arXiv preprint arXiv:1610.02136* (2016).

[26] Philipp Hennig and Søren Hauberg. “Probabilistic solutions to differential equations and their application to Riemannian statistics”. In: *Artificial Intelligence and Statistics*. 2014, pp. 347–355.

[27] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. “Neural networks for machine learning lecture 6a overview of mini-batch gradient descent”. In: *Cited on* 14.8 (2012).

[28] Johan Hoffman et al. “Turbulent Flow and Fluid–structure Interaction”. In: *Automated Solution of Differential Equations by the Finite Element Method, Volume 84 of Lecture Notes in Computational Science and Engineering*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Springer, 2012. Chap. 28.

[29] Johan Hoffman et al. “Unicorn: Parallel Adaptive Finite Element Simulation of Turbulent Flow and Fluid-Structure Interaction for Deforming Domains and Complex Geometry”. In: *Computer and Fluids* in press (2012).

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

[31] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks”. In: *Neural networks* 3.5 (1990), pp. 551–560.

[32] Raphael Hunger. *Floating point operations in matrix-vector calculus*. Munich University of Technology, Inst. for Circuit Theory and Signal ..., 2005.

[33] Bunpei Irie and Sei Miyake. “Capabilities of three-layered perceptrons.” In: *ICNN*. 1988, pp. 641–648.

[34] Niclas Jansson, Johan Jansson, and Johan Hoffman. “Framework for Massively Parallel Adaptive Finite Element Computational Fluid Dynamics on Tetrahedral Meshes”. In: *SIAM Journal on Scientific Computing* 34.1 (2012), pp. C24–C41.

[35] Alex Kendall and Yarin Gal. “What uncertainties do we need in bayesian deep learning for computer vision?” In: *Advances in neural information processing systems*. 2017, pp. 5574–5584.

[36] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[37] Diederik P Kingma and Max Welling. “Auto-encoding variational bayes”. In: *arXiv preprint arXiv:1312.6114* (2013).

[38] Robert C. Kirby. “Algorithm 839: FIAT, a New Paradigm for Computing Finite Element Basis Functions”. In: *ACM Transactions on Mathematical Software* 30.4 (2004), pp. 502–516. DOI: 10.1145/1039813.1039820.

[39] Robert C. Kirby. “FIAT: Numerical Construction of Finite Element Basis Functions,” in: *Automated Solution of Differential Equations by the Finite Element Method, Volume 84 of Lecture Notes in Computational Science and Engineering*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Springer, 2012. Chap. 13.

[40] Robert C. Kirby and Anders Logg. “A Compiler for Variational Forms”. In: *ACM Transactions on Mathematical Software* 32.3 (2006). DOI: 10.1145/1163641.1163644.

[41] Robert C. Kirby and Anders Logg. “Benchmarking Domain-Specific Compiler Optimizations for Variational Forms”. In: *ACM Transactions on Mathematical Software* 35.2 (2008), pp. 1–18. DOI: 10.1145/1377612.1377614.

[42] Robert C. Kirby and Anders Logg. “Efficient Compilation of a Class of Variational Forms”. In: *ACM Transactions on Mathematical Software* 33.3 (2007). DOI: 10.1145/1268769.1268771.

[43] Robert C. Kirby and L. Ridgway Scott. “Geometric Optimization of the Evaluation of Finite Element Matrices”. In: *SIAM Journal on Scientific Computing* 29.2 (2007), pp. 827–841.

[44] Robert C. Kirby et al. “Optimizing the Evaluation of Finite Element Matrices”. In: *SIAM Journal on Scientific Computing* 27.3 (2005), pp. 741–758. DOI: 10.1137/040607824.

[45] Robert C. Kirby et al. “Topological Optimization of the Evaluation of Finite Element Matrices”. In: *SIAM Journal on Scientific Computing* 28.1 (2006), pp. 224–240. DOI: 10.1137/050635547.

[46] Debasis Kundu. “Discriminating between normal and Laplace distributions”. In: *Advances in Ranking and Selection, Multiple Comparisons, and Reliability*. Springer, 2005, pp. 65–79.

[47] Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. “Artificial neural networks for solving ordinary and partial differential equations”. In: *IEEE transactions on neural networks* 9.5 (1998), pp. 987–1000.

[48] Isaac E Lagaris, Aristidis C Likas, and Dimitris G Papageorgiou. “Neural-network methods for boundary value problems with irregular boundaries”. In: *IEEE Transactions on Neural Networks* 11.5 (2000), pp. 1041–1049.

[49] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. “Simple and scalable predictive uncertainty estimation using deep ensembles”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 6402–6413.

[50] Alan Lapedes and Robert Farber. “How neural nets work”. In: *Evolution, learning and cognition*. World Scientific, 1988, pp. 331–346.

[51] Stig Larsson and Vidar Thomée. *Partial differential equations with numerical methods*. Vol. 45. Springer, 2003.

[52] Yann Le Cun and Françoise Fogelman-Soulié. “Modèles connexionnistes de l’apprentissage”. In: *Intellectica* 2.1 (1987), pp. 114–143.

[53] Hyuk Lee and In Seok Kang. “Neural algorithm for solving differential equations”. In: *Journal of Computational Physics* 91.1 (1990), pp. 110–131.

[54] Anders Logg. “Automating the Finite Element Method”. In: *Archives of Computational Methods in Engineering* 14.2 (2007), pp. 93–138. DOI: 10.1007/s11831-007-9003-9.

[55] Anders Logg. “Efficient Representation of Computational Meshes”. In: *International Journal of Computational Science and Engineering* 4.4 (2009), pp. 283–295. DOI: 10.1504/IJCSE.2009.029164.

[56] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

[57] Anders Logg and Garth N. Wells. “DOLFIN: Automated Finite Element Computing”. In: *ACM Transactions on Mathematical Software* 37.2 (2010). DOI: 10.1145/1731022.1731030.

[58] Anders Logg, Garth N. Wells, and Johan Hake. “DOLFIN: a C++/Python Finite Element Library”. In: *Automated Solution of Differential Equations by the Finite Element Method, Volume 84 of Lecture Notes in Computational Science and Engineering*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Springer, 2012. Chap. 10.

[59] Anders Logg et al. “FFC: the FEniCS Form Compiler”. In: *Automated Solution of Differential Equations by the Finite Element Method, Volume 84 of Lecture Notes in Computational Science and Engineering*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth N. Wells. Springer, 2012. Chap. 11.

[60] Lu Lu et al. “Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators”. In: *Nature Machine Intelligence* 3.3 (2021), pp. 218–229.

[61] A Malek and R Shekari Beidokhti. “Numerical solution for high order differential equations using a hybrid neural network—optimization method”. In: *Applied Mathematics and Computation* 183.1 (2006), pp. 260–271.

[62] Zhiping Mao et al. “DeepM&Mnet for hypersonics: Predicting the coupled flow and finite-rate chemistry behind a normal shock using neural-network approximation of operators”. In: *arXiv preprint arXiv:2011.03349* (2020).

[63] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[64] Yurii E Nesterov. “A method for solving the convex programming problem with convergence rate O (1/k^ 2)”. In: *Dokl. akad. nauk Sssr*. Vol. 269. 1983, pp. 543–547.

[65] Edward W Ng and Murray Geller. “A table of integrals of the error functions”. In: *Journal of Research of the National Bureau of Standards B* 73.1 (1969), pp. 1–20.

[66] Kristian B. Ølgaard, Anders Logg, and Garth N. Wells. “Automated Code Generation for Discontinuous Galerkin Methods”. In: *SIAM Journal on Scientific Computing* 31.2 (2008), pp. 849–864. DOI: 10.1137/070710032.

[67] Kristian B. Ølgaard and Garth N. Wells. “Optimisations for Quadrature Representations of Finite Element Tensors Through Automated Code Generation”. In: *ACM Transactions on Mathematical Software* 37 (2010). DOI: 10.1145/1644001.1644009.

[68] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. “Physics Informed Deep Learning (Part I): Data-driven solutions of nonlinear partial differential equations”. In: *arXiv preprint arXiv:1711.10561* (2017).

[69] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. “Physics informed deep learning (Part II): data-driven discovery of nonlinear partial differential equations”. In: *arXiv preprint arXiv:1711.10566* (2017).

[70] Prajit Ramachandran, Barret Zoph, and Quoc V Le. “Searching for activation functions”. In: *arXiv preprint arXiv:1710.05941* (2017).

[71] Carl Edward Rasmussen. "Gaussian processes in machine learning". In: *Summer school on machine learning*. Springer. 2003, pp. 63–71.

[72] Marie E. Rognes, Robert C. Kirby, and Anders Logg. "Efficient Assembly of H(div) and H(curl) Conforming Finite Elements". In: *SIAM Journal on Scientific Computing* 31.6 (2009), pp. 4130–4151. DOI: 10.1137/08073901X.

[73] Marie E. Rognes et al. "Automating the solution of PDEs on the sphere and other manifolds in FEniCS 1.2". In: *Geoscientific Model Development* 6 (2013), pp. 2099–2119. DOI: 10.5194/gmd-6-2099-2013.

[74] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

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

[76] Justin Sirignano and Konstantinos Spiliopoulos. "DGM: A deep learning algorithm for solving partial differential equations". In: *Journal of Computational Physics* 375 (2018), pp. 1339–1364.

[77] Christian Szegedy et al. "Rethinking the inception architecture for computer vision". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.

[78] Jun Takeuchi and Yukio Kosugi. "Neural network representation of finite element method". In: *Neural Networks* 7.2 (1994), pp. 389–395.

[79] Rohit Tripathy and Ilias Bilionis. "Deep UQ: Learning deep neural network surrogate models for high dimensional uncertainty quantification". In: *Journal of Computational Physics* 375 (2018), pp. 565–588.

[80] Sifan Wang, Hanwen Wang, and Paris Perdikaris. "Learning the solution operator of parametric partial differential equations with physics-informed DeepOnets". In: *arXiv preprint arXiv:2103.10974* (2021).

[81] E Weinan, Jiequn Han, and Arnulf Jentzen. "Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations". In: *Communications in Mathematics and Statistics* 5.4 (2017), pp. 349–380.

[82] Nick Winovich, Karthik Ramani, and Guang Lin. “ConvPDE-UQ: Convolutional neural networks with quantified uncertainty for heterogeneous elliptic partial differential equations on varied domains”. In: *Journal of Computational Physics* 394 (2019), pp. 263–279.

[83] Yinhao Zhu and Nicholas Zabaras. “Bayesian deep convolutional encoder–decoder networks for surrogate modeling and uncertainty quantification”. In: *Journal of Computational Physics* 366 (2018), pp. 415–447.

VITA

Nick Winovich graduated from the University of Notre Dame in 2012, majoring in mathematics and Spanish, and received funding from the Kellog Institute for International Studies to teach English at a primary school in Pacuare, Costa Rica through the World Teach program. He subsequently received a master's degree in mathematics at the University of Oregon and enrolled as a Ph.D. student in the Department of Mathematics at Purdue University in 2015. At Purdue, he was funded as an NSF IGERT fellow as a part of the sustainable electronics program headed by Dr. Carol Handwerker and later participated in a student internship at Sandia National Laboratories conducting research under the supervision of Dr. Mohamed Ebeida. He also volunteers as a student consultant for the Purdue Data Science Consulting Service and has served as a mentor for undergraduate students through the NSF Summer Undergraduate Research Fellowship (SURF) and Network for Computational Nanotechnology (NCN) Undergraduate Research Experience (URE) programs.

PUBLICATIONS

Winovich, N., Ramani, K., & Lin, G. (2019). ConvPDE-UQ: Convolutional neural networks with quantified uncertainty for heterogeneous elliptic partial differential equations on varied domains. Journal of Computational Physics, 394, 263-279.

Winovich, N., Rushdi, A., Phipps, E. T., Ray, J., Lin, G., & Ebeida, M. S. (2019). Rigorous Data Fusion for Computationally Expensive Simulations (No. SAND2019-10322). Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); Sandia National Laboratories, Livermore, CA.

Kim, M., Winovich, N., Lin, G., & Jeong, W. (2019). Peri-Net: Analysis of Crack Patterns Using Deep Neural Networks. Journal of Peridynamics and Nonlocal Modeling, 1(2), 131-142.

Kim, S., Winovich, N., Chi, H. G., Lin, G., & Ramani, K. (2019). Latent transformations neural network for object view synthesis. The Visual Computer, 1-15.