University of Cincinnati

Date: 9/15/2022

I, Mohammed Barakat M Alamari, hereby submit this original work as part of the requirements for the degree of Doctor of Philosophy in Mathematical Sciences.

It is entitled:

Neural Network Emulation for Computer Model with High Dimensional Outputs using Feature Engineering and Data Augmentation

Student's name: **Mohammed Barakat M Alamari**

This work and its defense approved by:

Committee chair: Won Chang, Ph.D.

Committee member: Emily Kang, Ph.D.

Committee member: Xia Wang, Ph.D.

44127

Neural Network Emulation for Computer Model with High Dimensional Outputs using Feature Engineering and Data Augmentation

A dissertation submitted to the

Graduate School

of the University of Cincinnati

in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

in the Department of Mathematical Sciences

of the College of Arts and Sciences

by

Mohammed Barakat M Alamari

Committee:

Won Chang, Ph.D., Chair.

Emily Kang, Ph.D.

Xia Wang, Ph.D.

Abstract

Expensive computer models (simulators) are frequently used to simulate the behavior of a complex system in many scientific fields because an explicit experiment is very expensive or dangerous to conduct. Usually, only a limited number of computer runs are available due to limited sources. Therefore, one desires to use the available runs to construct an inexpensive statistical model, an emulator. Then the constructed statistical model can be used as a surrogate for the computer model. Building an emulator for high dimensional outputs with the existing standard method, the Gaussian process model, can be computationally infeasible because it has a cubic computational complexity that scales with the total number of observations. Also, it is common to impose restrictions on the covariance matrix of the Gaussian process model to keep computations tractable. This work constructs a flexible emulator based on a deep neural network (DNN) with feedforward multilayer perceptrons (MLP). High dimensional outputs and limited runs can pose considerable challenges to DNN in learning a complex computer model's behavior. To overcome this challenge, we take advantage of the computer model's spatial structure to engineer features at each spatial location and then make the training of DNN feasible. Also, to improve the predictive performance and avoid overfitting, we adopt a data augmentation technique into our method. Finally, we apply our approach using data from the UVic ESCM model and the PSU3D-ICE model to demonstrate good predictive performance and compare it with an existing state-of-art emulation method.

Acknowledgements

Words cannot express my deepest gratitude to my advisor Prof. Won Chang for the tremendous help, clear guidance, and continued encouragement. I am fortunate to have the opportunity to work with Prof. Won Chang; without his expertise, patience, and insightful comments, this work would not be possible. I am incredibly thankful to my dissertation committee members, Prof. Xia Wang and Prof. Emily Kang, for giving up their valuable time to serve on my dissertation committee and for their insightful and helpful remarks that greatly improved this work.

Many thanks to all Professors I had as instructors at the University of Cincinnati. I would like to extend my thanks to all Professors and staff members in the Department of Mathematical Sciences at the University of Cincinnati. Also, I am grateful to King Khalid University, Saudi Arabia, for providing financial support to pursue graduate studies in the United States.

I would like to thank my beloved wife for being a great supporter and continuing to believe in me through tough times. Without her help, sacrifices, and patience, achieving this goal would have been much more difficult. To my precious children, thank you for bringing so much joy and happiness to my life. To my dear Mother, I am deeply grateful for all the unconditional love and support. To my amazing siblings, thank you for your encouragement and motivation.

DEDICATION

I dedicate this dissertation work to my father, in loving memory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer models (simulators) are frequently used to simulate the behavior of a complex physical system in many scientific fields because an explicit experiment is costly, dangerous, or even impossible to conduct. Computer models are now commonplace in many scientific fields, such as engineering, biological, and climate science. For example, in climate science, computer models are built to mimic the actual world so scientists can understand the past and project climate scenarios in the future. Typically, computer models represent physical phenomena with a complex system of mathematical equations according to laws of physics and mathematical theories. Scientists can only solve them using sophisticated computers (Smith (2013); Gramacy (2020)).

Computer models have a complex structure and are expensive to run, and could take hours, days, or even weeks to obtain a single run at a specific input setting. Still, they are not perfect because they represent an ideal reality of the world, and they cannot have all the physical processes we need in them due to ignorance or computation limits. Consequently, these computer models have many sources of uncertainty, and these sources have to be quantitatively characterized and reduced to produce a reliable simulation of the target physical system. Identifying and reducing uncertainties related to computer models falls under a research field called uncertainty quantification (UQ). Uncertainty quantification (UQ) is an interdisciplinary field of applied science and has become increasingly popular due to the tremendous use of computer models (computer experiments) in modern science. Since uncertainty quantification (UQ) problems such as uncertainty propagation, computer model calibration, and sensitivity analysis require a considerable number of runs of the computer models, building an inexpensive statistical model is desirable to cope with the expensive computer models (Santner et al. (2003); Gramacy (2020)).

Sacks et al. (1989) were among the first to propose modeling the computer model's output as a Gaussian process (GP) realization. The Gaussian process model is also known as Kriging in geostatistics, where the input space's dimension is typically two (latitude and longitude) or at most three (Cressie (1993)). The characteristics of the data generated by computer models make them different from data in other applications, such as data in spatial statistics (Gu (2016); Gramacy (2020)). Therefore, in the procedure of building an emulator, we need to consider these characteristics.

First, It is common for computer models to be deterministic. That is, the repeated runs of the computer model at the same input parameter settings will always generate the same outputs. In this work, we consider deterministic climate computer models. Hence, building an emulator that nearly interpolates the computer model outputs in the input parameter space is desirable, or at least it is appropriate (Sacks et al. (1989); Gramacy (2020)). Gaussian process can be an exact interpolator by setting the so-called nugget term (jitter) to zero. However, a zero nugget term is not recommended for better numerical stability (Neal (1997); Gramacy and Lee (2008); Pepelyshev (2010)) and better predictive accuracy and coverage when data are sparse or the Gaussian process model assumptions are not satisfied (e.g., stationarity) (Gramacy and Lee (2012)). Second, the input parameters of the computer models are usually on different scales and have different units. Hence we expect the correlation of the outputs would not decay homogeneously in each input parameter direction; this property is called anisotropy, as opposed to isotropy. In comparison with spatial statistics setting, an isotropic correlation structure (e.g., isotropic Gaussian, Matérn, etc.) is often appropriate since the axes (e.g., longitude and latitude) are on the same scale (Ugarte (2015); Gu (2016)). Gaussian process can maneuver the anisotropy by allowing different correlation rates in each input direction. In practice, this is not an easy job; and often, a separable correlation structure (e.g., anisotropic Gaussian) is used in computer models for computational purposes (Ugarte (2015); Gu (2016); Gramacy (2020)). Third, the relationship between the computer model outputs and input parameters is highly nonlinear with complicated interactions; a flexible statistical model is required to approximate the computer model sufficiently.

Although the Gaussian process is a powerful and standard method to construct a statistical model that interpolates computer model outputs in the computer model parameter space, it has limitations. The primary limitation of the Gaussian process on large-scale applications is evaluating an expensive likelihood which involves finding the inversion of a large dense covariance matrix. Typically, the time complexity of computing a likelihood with n-dimensional outputs at r different parameter settings requires $O(n^3 r^3)$ operations. Therefore, the Gaussian process model can be computationally infeasible for large-scale problems in computer model literature.

A considerable amount of research has been done to improve the time computation of the Gaussian process model in the literature. Various approaches are possible, primarily relying on imposing certain constraints on the covariance structure. Defining a separable covariance structure is a common approach; as a result, the large covariance matrix can be decomposed as a Kronecker product of much smaller matrices, and then one can exploit the Kronecker structure to speed up the computation (Rougier (2008); Conti and O'Hagan (2010)).

Dimension reduction techniques are another natural approach for overcoming the high dimensionality issue, where one finds proper orthogonal basis representations of high dimensional outputs into a lower-dimensional space. Then the reduced and transformed outputs can be modeled as independent Gaussian processes (e.g., principal components (Higdon et al. (2008); Chang et al. (2014)) or wavelet decomposition (Bayarri et al. (2007)).

Yet another approach would be low-rank methods which approximate the Gaussian process covariance matrix with a lower-dimensional structure. Then, the computation saving is achieved by replacing the expensive covariance matrix with a low–rank approximation that has a computationally tractable form, i.e., the low-rank methods utilize the Sherman-Woodbury-Morrison structure in the calculation to exploit the low-rank approximation (e.g., Fixed rank kriging (Cressie and Johannesson (2008)), Gaussian predictive processes (Banerjee et al. (2008); Finley et al. (2009))). Other approaches reduce the computational burden by inducing sparsity in the covariance matrix, e.g., covariance tapering (Kaufman et al. (2008)) or in the precision matrix using the assumption of conditional independence, e.g., Nearest-Neighbor Gaussian Process (NNGP) (Datta et al. (2016)). Then, sparsity structure can be exploited in the calculation.

Emulating computer model outputs is a well-studied subject; however, there are considerable challenges to building an emulator for computer models with high dimensional outputs. The high dimensional outputs are usually in the form of spatial fields with a complicated dependence structure. Unfortunately, most existing emulation procedures based on Gaussian processes have difficulty scaling well with high dimensional outputs of complex computer models due to restrictive assumptions such as a separable covariance matrix structure or a particular parametric form of the covariance matrix. Although such assumptions are computationally convenient, this can result in poor performance of the emulator of complicated computer models. It is far less common to construct an emulator that copes well with high dimensional outputs, such as (Gu and Berger (2016)), than for low dimensional problems in the computer emulation literature. Nowadays, it is increasingly common in many scientific fields to develop a highly sophisticated computer model with very large outputs due to recent technological advances and computing power. Developing an emulator with sufficient flexibility to approximate a complicated computer model with high dimensional outputs is an active area of research in many fields of science and engineering.

In recent years, deep neural networks have revolutionized the artificial intelligence (AI) field and reached state-of-the-art results in many complex problems in the AI field, such as image recognition, language processing, speech recognition, and many more (Krizhevsky et al. (2012); Graves et al. (2013); Deng et al. (2013); Goodfellow et al. (2016)). Many fundamentals of deep neural networks were developed decades ago; however, deep neural networks did not become popular until recent years. The recent advancement and popularity of deep neural networks have flourished by many factors, such as an increase in computing power, an increase in available training data (big data), recent advancements in stochastic gradient-based optimizations, and the development of practical-open-source software libraries for deep learning (Goodfellow et al. (2016); Tripathy and Bilionis (2018)).

A wide range of applications has shown that deep neural networks are highly flexible and powerful enough to model complex nonlinear relationships (Goodfellow et al. (2016)) and some theoretical aspects of deep learning's powerful nonlinear function approximation have been investigated (e.g., Poggio et al. (2017); Chen et al. (2019)). The recent advancements and successes of deep neural networks have motivated many researchers in computer model literature and related fields to explore opportunities for deep learning as an alternative paradigm to model data with complicated structures (e.g., Bhatnagar et al. (2020); Wang et al. (2019); Tripathy and Bilionis (2018); Di et al. (2016)).

1.1 Overview of the Proposed Method

Inspired by the recent improvements and successes of deep neural networks, we propose in this work an alternative approach for constructing a flexible emulator of computer models with high dimensional outputs and limited runs using a deep neural network with a feedforward multilayer perceptron (MLP).

In computer model emulation problems, we often face a limited number of runs. Thus using deep neural networks to build emulators can be significantly challenging to train the neural networks to learn the underlying system of computer models accurately. This issue is further complicated by the presence of high dimensional outputs. Therefore, we introduce an approach to engineer input features for neural networks to make the training more practical under constraints of high dimensional outputs and limited runs. We call this way of finding input features for the neural network parallel features engineering.

Computer model outputs are highly correlated with complex dependence structures. Incorporating information from neighboring outputs can improve the constructed emulator's prediction accuracy. We use the proposed parallel features engineering to create several sets of input features for neural networks to incorporate the neighboring information in the parameter and spatial space. The input features include parameter settings, spatial coordinates, and neighboring information in the parameter and spatial space.

Also, to substantially improve the predictive performance of neural networks, we adopt a data augmentation technique into our approach. Our data augmentation will increase the size and diversity of the training data points. The proposed data augmentation will enable us to include outputs' information of various ranges instead of only including short-range outputs' information and thus enhance the learning dynamic of neural networks.

1.2 Outline of the Dissertation

We have organized the rest of the dissertation as follows: Chapter 2 provides a brief overview of computer models and presents the emulation problem. In Chapter 3, we discuss deep neural networks (DNN) in general, and we review the properties of a neural network architecture related to our work, regularization techniques in deep learning literature, and the parameters optimization of neural networks. In Chapter 4, we start with descriptions of the data from the computer models we used in this work. Then Chapter 4 explains the dimensionality problem of training neural networks with high dimensional output and limited runs. Also, Chapter 4 develops a features engineering approach to overcome the dimensionality problem and adapt a data augmentation technique to improve the prediction accuracy of our emulator. Chapter 5 designs numerical studies that investigate the effectiveness of the proposed data augmentation method on the prediction performance under different scenarios. In addition, Chapter 5 shows that our method outperforms an existing state-of-art method under different situations. In Chapter 6, we summarize the contributions and findings of our work and report the conclusions of this dissertation.

Chapter 2

Computer Models and Uncertainty Quantification

Over the last decades, computer models have become indispensable tools for simulating complicated phenomena such as weather models, climate models, and biological models. Computer models enable us to study complex phenomena and solve difficult problems through computer experiments (simulations); otherwise, it would be expensive, dangerous, or impossible to conduct physical experiments. Typically, computer models are sophisticated codes implemented in computers to solve a highly complex system of partial differential equations using numerical methods such as the finite element method. Computer models take a set of input parameters that are believed to govern the process of interest and return a quantity of interest (output or response).

Computer models are often deterministic, meaning that the runs of computer code at the same input parameter settings will always produce the same outputs. Although computer models can have a stochastic nature, this work's emphasis will be on computer models of deterministic nature.

It is often that computer models are treated as black-box functions. Many scenarios can lead researchers to treat computer models as black boxes, such as intractable closed-form solutions of the underlying mathematical equations or restricted access to internal aspects of computer models. Consequently, running computer models is required to reveal any information about the underlying solution at specific parameter settings; the parameter settings can represent a particular situation of interest. Herein we think of the computer models as a black-box that takes inputs and returns outputs of the simulated process. **Figure 2.1** demonstrates the black-box treatment of computer models, in which only the inputs and outputs of the simulated process are visible.

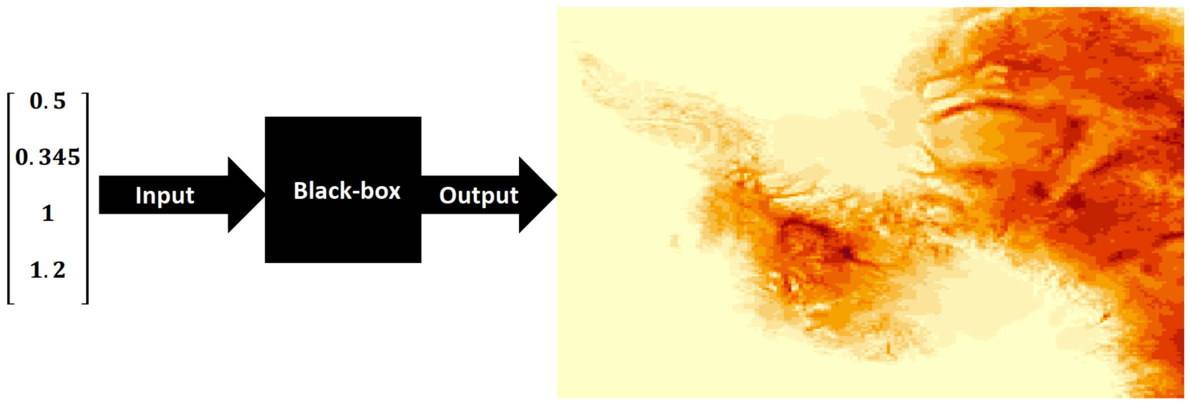


Figure 2.1: *Computer models as a black-box*

Although Computer models are essential for gaining insight into the behavior of target complex systems, they are typically time-consuming and expensive for tasks requiring hundreds or thousands of runs, such as uncertainty quantification tasks.

2.1 Uncertainty Quantification Tasks

Computer models are excellent tools to simulate complicated systems but are not perfect. Real-world processes are inherently chaotic and complex. Computer models can not represent real-world phenomena perfectly or include all related physical processes due to a lack of knowledge, limited computing power, and budget constraint. Uncertainty quantification is a cross-disciplinary field that aims to identify and reduce uncertainties related to computer models to produce reliable simulations. Typical uncertainty quantification tasks of interest include but are not limited to computer model calibration, uncertainty propagation, optimization, sensitivity analysis, and computer model discrepancy.

2.1.1 Computer Model Calibration

One primary source of uncertainty associated with computer models is uncertainties about the computer model's parameters' actual values. Typically, the true values of some of the input parameters that influence the outputs of computer models are unknown. These input parameters are known as calibration parameters, and their actual value needs to be estimated to make valid inferences about the actual process under study. The formal approach that estimates the calibration parameters by utilizing information from available actual observations of the process and computer model runs is called computer model calibration, also known as the inverse problem (Chang et al. (2014)).

2.1.2 Uncertainty Propagation

As we mentioned earlier, computer models are not perfect, and they have many sources of uncertainties. One popular task in uncertainty quantification literature is uncertainty propagation, also called forward uncertainty quantification. Uncertainty propagation is the procedure of quantifying uncertainties in computer model outputs that are propagated from uncertainties in input parameters (Tripathy and Bilionis (2018)).

2.1.3 Optimization

Computer models are used extensively for decision-making in many scientific and engineering problems, such as design issues with helicopter rotor blades (Booker et al. (1999)). The design problems can be formulated as an optimization problem where the goal is to search for optimal input design variables. After solving the optimization problem, informed decisions about the design variables can be made.

2.1.4 Sensitivity Analysis

We can define sensitivity analysis as the procedure that investigates how the computer model outputs respond to changes in the input parameters. Sensitivity analysis is a priceless tool for a process where it is of great importance to know which are the input parameters that mainly affect the variability of the output (Oakley and O'Hagan (2004)).

2.1.5 Data Models Discrepancy

The difference between the computer model outputs and actual observations of the process of interest is called data model discrepancy—the discrepancy results from the fact that computer models are imperfect and rely on assumptions and simplifications that do not hold in reality. Recognizing model discrepancies is critical for making appropriate use of the observations of the process of interest. The model discrepancy is also called by other names, such as model inadequacy (Kennedy and O'Hagan (2001)).

2.2 Emulation Problem

Uncertainty quantification tasks require hundreds or thousands or even tens of thousands of repeated evaluations of computer models. Consequently, performing uncertainty quantification tasks can be highly challenging due to the high computational power that is required for each evaluation of the computer models or the long wait time it takes to complete each evaluation of the computer models. For example, if each computer code evaluation takes an hour to complete, 1000 evaluations will take 42 days.

As a result of the challenges mentioned above, building a low-cost statistical model, an emulator, to replace the expensive and computationally time-consuming computer model is necessary to perform hundreds or thousands of runs to properly quantify the uncertainties associated with the computer model. The problem of constructing an inexpensive approximation of computer models is called computer emulation. So, an emulator can be defined as a cheap approximation of an expensive computer model. Emulators and surrogates are frequently used interchangeably to mean the same thing.

To formulate the emulation problem, we take the opportunity to establish a set of notations that will be used throughout this thesis. The computer model is a black-box function $\boldsymbol{\eta}$ that takes a set of input parameters $\boldsymbol{\theta_i}$ from the input space $\boldsymbol{\Theta} \subseteq \mathbb{R}^v$ and returns an output $\mathbf{Y_i}$ of n-dimensional space. Typically, the computer model simulates the quantity of interest over a known fixed grid of spatial locations $\{\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_j, ..., \mathbf{s}_n\} = \mathcal{S} \subseteq \mathbb{R}^d$, such as meshes used in numerical methods for solving partial differential equations like finite-difference methods, finite element methods, and finite-volume methods. It is worth noting that $\mathcal{S}$ is no longer variable since it is known and fixed; thus, we loosely represent the computer model as a function mapping from the input parameter space to the n-dimensional space.

$$\begin{aligned} \boldsymbol{f} :& \boldsymbol{\Theta} \to \mathbb{R}^n \\ & \boldsymbol{\theta_i} \to \mathbf{Y_i} \end{aligned} \tag{2.1}$$

The corresponding output of evaluating the computer model at a specific parameter setting $\boldsymbol{\theta_i}$ is a spatial process that is computed at n known different spatial locations $\mathbf{Y_i} = (Y(\boldsymbol{\theta}_i, \mathbf{s}_1), Y(\boldsymbol{\theta}_i, \mathbf{s}_2), ...,$ $Y(\boldsymbol{\theta}_i, \mathbf{s}_j), ..., Y(\boldsymbol{\theta}_i, \mathbf{s}_n))^T$.

Typically, emulators are constructed with limited number of runs due to limited resources. Therefore, at r carefully chosen design points $\boldsymbol{\theta_i} \in \boldsymbol{\Theta} \subseteq \mathbb{R}^v, i = 1, 2, ..., r$, the computer model is executed. The r design points and their corresponding computer model outputs $\{\boldsymbol{\theta}_i, \mathbf{Y_i}\}_i^r$ are often referred to as the training data. Then we use the available model runs to construct an emulator $\hat{\boldsymbol{f}}$ that mimics the input-output functional relationship in the black-box computer model. After building the emulator $\hat{\boldsymbol{f}}$, we can use it to predict the computer model outputs at untried parameter settings $\boldsymbol{\theta}_i^*$ and perform uncertainty quantification tasks such as computer model calibration, sensitivity analysis, optimization, and uncertainty propagation.

In the literature, there are a variety of approaches for creating an emulator, and it is a growing field of study. Methods that use Gaussian processes are widespread and popular for emulating the computer model outputs. For a more detailed review of the Gaussian process model and its properties and drawbacks, please see (Gramacy (2020)). In our work, we construct an emulator based on neural networks. Chapter 3 reviews neural networks and some of their properties that are of interest to our work.

Chapter 3

Deep Neural Networks

Neural networks were around many decades ago. The original idea of neural networks is to mimic some of the functionality of biological systems in the human brain (Block (1962)). This idea attracted researchers in the neuroscience, computer science, applied mathematics, and statistics communities to study the potential capabilities of neural networks. However, due to the limitations in training data and computing power at that time, neural networks did not achieve significant performance and became less popular.

Recent Increases in computing power, availability of vast training data, and other advancements have significantly improved the performance of neural networks and reached state-of-the-art performance in many challenging tasks. Since then, large neural networks trained with massive data have become very popular and among the top-performing algorithms in the machine learning literature.

The fundamental building component of a neural network is called a neuron since, in principle, neural networks mimic the human brain. Sometimes neurons are also called unites or nodes. A neuron takes inputs, performs a mathematical operation on the inputs, and then returns outputs.

Layers of neural networks consist of a stack of neurons. Neural networks begin with an input layer and end with an output layer. The term hidden layers refers to the intermediate layers. The number of neurons in a layer refers to the width or size of the layer. The depth of a neural network refers to the total number of layers in the network. The term architecture refers to the configuration of the network's layers and nodes.

When neural networks contain several hidden layers, they are called deep neural networks. Deep neural networks are a potent tool with excellent modeling capabilities in machine learning literature and have a very expressive architecture that allows them to capture complicated phenomena in the real world. Deep neural networks can accurately approximate an arbitrary input-output functional relationship given rich training data and sufficiently large neural network architecture.

There are many types of neural network architectures. However, we do not plan to give a comprehensive review of the neural network architectures. For a more thorough review, please see Goodfellow et al. (2016). Instead, we focus on one type of neural network architectures in the following section and review some of its properties related to our work.

3.1 Feedforward Neural Networks

One well-known architecture of deep neural networks is feedforward neural networks (FNNs). FNNs are also called feedforward multilayer perceptrons (MLPs) or neural networks (NNs). MLPs are universal function approximations under mild conditions, which is an appealing characteristic (Hornik et al. (1989)). In other words, It has been proven that sufficiently large neural network architectures with enough data can describe arbitrary complex functions.

MLPs are extremely important in deep learning. They function as the building blocks for many famous and specialized neural networks, such as Convolutional neural networks (CNN), Long short-term memory (LSTM), and many more (LeCun et al. (1995); Huang et al. (2015)). MLPs have a wide range of applications and, unlike CNN and LSTM, do not require inputs to be in specific formats. In this sense, MLPs can be seen as a more flexible architecture because it has broader applicability than the other networks (e.g., CNN or LSTM). In this manuscript, we use the names feedforward multilayer perceptrons, neural networks, and deep neural networks interchangeably.

3.1.1 Architecture of Feedforward Neural Networks

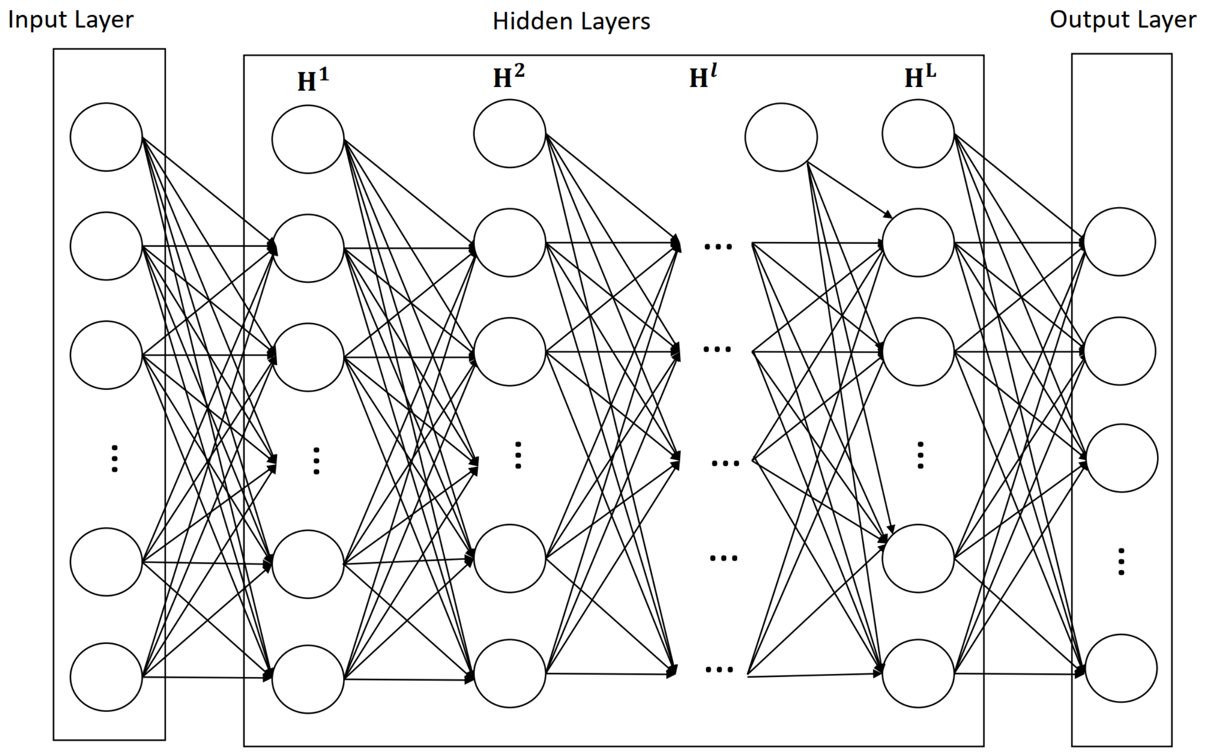


Figure 3.1: *Feedforward deep neural network architecture*

Figure 3.1 shows a general architecture of feedforward neural networks. These networks are called feedforward because information flows from the input layer to the output layer, and there is no feedback connection between the neurons. Typically, the width of the input layer is determined by the dimension of the input features plus one for the intercept term, while the dimension of the output determines the size of the output layer. The practitioners specify the widths of the hidden layers. We can increase the neural network models' flexibility by increasing the width of the hidden layers or the depth of the networks. For additional information about feedforward neural networks, please see (Goodfellow et al. (2016), Chapter 6).

To be more precise, consider an input vector $\mathbf{X}_i$ of dimension p with corresponding response $\mathbf{Y}_i \equiv \boldsymbol{f}_i$ of dimension n generated by an unknown function $\boldsymbol{f}(.)$. Let d_0 denotes the number of neurons in the input layers and d_l is the number of neurons in the l^{th} hidden layer. Also, assume d_{L+1} is the number of neurons in the output layer. We can express the architecture of feedforward neural networks using matrix notations as a hierarchical structure:

$$\begin{aligned}
\mathbf{H}^{(1)} =& g^{(1)}(\mathbf{W}^{(1)}\mathbf{X} + \mathbf{b}^{(1)}) \\
\mathbf{H}^{(2)} =& g^{(2)}(\mathbf{W}^{(2)}\mathbf{H}^{(1)} + \mathbf{b}^{(2)}) \\
& ... \\
\mathbf{H}^{(l)} =& g^{(l)}(\mathbf{W}^{(l)}\mathbf{H}^{(l-1)} + \mathbf{b}^{(l)}) \\
& ... \\
\mathbf{H}^{(L)} =& g^{(L)}(\mathbf{W}^{(L)}\mathbf{H}^{(L-1)} + \mathbf{b}^{(L)}) \\
\hat{\boldsymbol{f}} =& \mathbf{W}^{(L+1)}\mathbf{H}^{(L)} + \mathbf{b}^{(L+1)}
\end{aligned} \tag{3.1}$$

where $\mathbf{W}^{(l)}$, for $l \in \{1, 2, ..., L, L+1\}$, are $d_l \times d_{l-1}$ weight matrices; $\mathbf{b}^{(l)}$, for $l \in \{1, 2, ..., L, L+1\}$, are d_l-dimensional intercept vectors (commonly called "bias" in the deep learning community).

$g^l : \mathbb{R}^{d_l} \to \mathbb{R}^{d_l}$ is an element-wise activation function for the neurons in the l^{th} layer and $\mathbf{H}^{(l-1)}$ is a d_l dimension vector of the neurons in the l^{th} layer. So the neurons in the hidden layers take inputs from the previous layer. They then perform a weighted sum of the inputs and apply an element-wise nonlinear transformation according to a specified activation function. The outputs of one layer are passed to the subsequent layer as inputs. $\hat{\boldsymbol{f}}(.)$ is the approximation function constructed by the deep neural network of the unknown underlying function $\boldsymbol{f}(.)$.

Recent studies in neural network approximation theory literature have shown that deep neural networks (i.e., $L \gg 1$) tend to have a small approximation error than shallow networks (Liang and Srikant (2016); Poggio et al. (2017); Chen et al. (2019)). Training deep neural networks can be problematic due to many problems, such as saturation (vanishing gradient), overfitting issues, and efficient optimization techniques. The proceeding sections review recent advancements in the deep network literature and how these advancements can help overcome these issues.

3.1.2 Model Fitting in Deep Neural Networks

The neural network structure $\hat{\boldsymbol{f}}$ in (3.1) can be fully described by its weights matrices $\mathbf{W}^{(l)}$ and biases vectors $\mathbf{b}^{(l)}$, where $l \in \{1, 2, ..., L, L+1\}$. Jointly, they are referred to as the network's parameters. Let $\boldsymbol{\omega}$ be the collection of the neural network's parameters, i.e., $\boldsymbol{\omega} = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L+1}$. Consequently, $\hat{\boldsymbol{f}}$ is a function in $\boldsymbol{\omega}$. To fit $\hat{\boldsymbol{f}}$ to a response of interest $\boldsymbol{f}$, we need to estimate the network's parameters $\boldsymbol{\omega}$. The parameters estimation problem can be treated as a minimization problem of an appropriate loss function $\mathcal{L}(\boldsymbol{\omega}; \boldsymbol{f})$ that captures the discrepancy between the true function $\boldsymbol{f}$ and the approximated function $\hat{\boldsymbol{f}}$. Eq. 3.2 formulates the parameters estimation problems as a minimization problem.

$$\hat{\boldsymbol{\omega}} = \arg\min_{\boldsymbol{\omega}} \mathcal{L}(\boldsymbol{\omega}; \boldsymbol{f}) \tag{3.2}$$

where $\hat{\boldsymbol{\omega}}$ are the estimates of neural network's parameters. In practice, we estimate $\boldsymbol{\omega}$ based on a finite set of data points, known as the training dataset, since we only observe $\boldsymbol{f}$ at finite inputs $\mathbf{Y}_1, \mathbf{Y}_2, ..., \mathbf{Y}_i, ..., \mathbf{Y}_N$. Let $\mathcal{D}_{train} = \{\mathbf{X}_i, \mathbf{Y}_i\}_{i=1}^{N_{train}}$ be the set of the training data of N_{train} data point examples. Then the problem of estimating the neural network's parameters, called the training or learning stage in deep neural literature, is reduced to the problem of minimizing the mismatch between the actual output $\mathbf{Y}_i$ and the predicted output $\hat{\mathbf{Y}}_i = \hat{\boldsymbol{f}}_i$ based on an appropriate loss function. The standard loss function for regression tasks (continuous response) is the mean squared loss function in deep neural literature.

$$\mathcal{L}(\boldsymbol{\omega}; \mathcal{D}_{train}) = \frac{1}{N_{train}} \sum_{i}^{N_{train}} (\hat{\mathbf{Y}}_i - \mathbf{Y}_i)^T (\hat{\mathbf{Y}}_i - \mathbf{Y}_i) \tag{3.3}$$

Where $\hat{\mathbf{Y}}_i$ is the prediction of the true output $\mathbf{Y}_i$. So, the objective is to find $\hat{\boldsymbol{\omega}}$ such that the deep neural network approximates $\mathbf{Y}_i$ satisfactory for a given input $\mathbf{X}_i$. The neural network is trained by minimizing the loss function in 3.3, which is equivalent to maximizing the log-likelihood function for the model in 3.4 under the Gaussian noise assumption.

$$\begin{aligned} \mathbf{Y}_i &= \hat{\boldsymbol{f}}(\mathbf{X}_i) + \boldsymbol{\epsilon}_i \\ &= \hat{\mathbf{Y}}_i + \boldsymbol{\epsilon}_i \end{aligned} \tag{3.4}$$

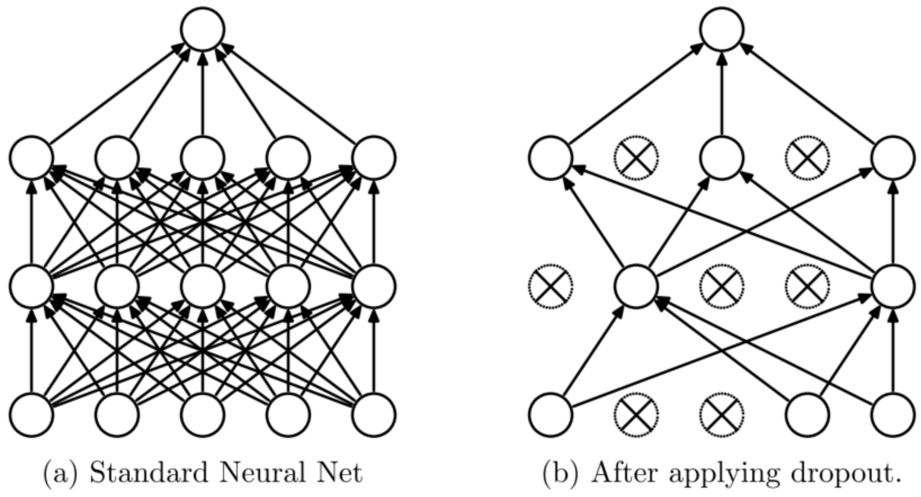
where $\boldsymbol{\epsilon}_i$ is an n-dimensional prediction error term and $\hat{\boldsymbol{f}}$ is the deep neural network, as we mentioned earlier.

3.1.3 Regularization techniques in deep learning

Deep neural networks are incredibly powerful tools with excellent modeling capabilities; however, they are prone to overfitting, typically leading to poor prediction performance (Srivastava et al. (2014)). Imposing regularization techniques often improve prediction performance.

3.1.3.1 Dropout

Dropout is a popular technique used in deep neural networks to avoid overfitting, which was proposed by Srivastava et al. (2014). The underlying idea of dropout is to introduce stochasticity in the likelihood function in model 3.4 by randomly dropping some nodes in the network architecture each time the likelihood is evaluated. **Figure 3.2** visually illustrates the basic idea of this regularization method.



Source: Srivastave et al., 2014

Figure 3.2: *Left: Neural network without dropout technique. Right: Neural network with dropout technique*

As a result, the network structure given in 3.1 needs to be modified, as shown below:

$$\begin{aligned}
\mathbf{H}^{(1)} =& g^{(1)}(\mathbf{W}^{(1)}\mathbf{X} * \mathbf{r}^{(1)} + \mathbf{b}^{(1)}) \\
\mathbf{H}^{(2)} =& g^{(2)}(\mathbf{W}^{(2)}\mathbf{H}^{(1)} * \mathbf{r}^{(2)} + \mathbf{b}^{(2)}) \\
&... \\
\mathbf{H}^{(l)} =& g^{(l)}(\mathbf{W}^{(l)}\mathbf{H}^{(l-1)} * \mathbf{r}^{(l)} + \mathbf{b}^{(l)}) \\
&... \\
\mathbf{H}^{(L)} =& g^{(L)}(\mathbf{W}^{(L)}\mathbf{H}^{(L-1)} * \mathbf{r}^{(L)} + \mathbf{b}^{(L)}) \\
\hat{\boldsymbol{f}} =& \mathbf{W}^{(L+1)}\mathbf{H}^{(L)} * \mathbf{r}^{(L+1)} + \mathbf{b}^{(L+1)}
\end{aligned} \tag{3.5}$$

where $*$ is the elementwise multiplication and $\mathbf{r}^{(l)}$ for $l = 1, ..., L+1$ is a d_l-dimensional random vector whose elements $\mathbf{r}^{(l)} = [\mathrm{r}_1^{(l)}, ..., \mathrm{r}_{d_l}^{(l)}]^T$ are identically and independently distributed Bernoulli random variables with a pre-specified success probability p_{keep}, i.e., $\mathrm{r}_1^{(l)}, ..., \mathrm{r}_{d_l}^{(l)} \sim$ iid. Bernoulli(p_{keep}). So,

$$\mathrm{r}_i^{(l)} = \begin{cases} 1 & \text{with probability } p_{keep} \\ 0 & \text{with probability } 1 - p_{keep} \end{cases}$$

where $i = 1, 2, ..., d_l$. This results in a stochastic loss function since every evaluation of the loss function depends on the random vectors, $\mathbf{r}^{(l)}$ for $l = 1, ..., L+1$. To emphasize the stochasticity of the loss function, we add $\mathbf{r}$ as a subscript to the loss function as the following:

$$\mathcal{L}_{\mathrm{r}}(\boldsymbol{\omega}; \mathcal{D}_{train}) = \frac{1}{N_{train}} \sum_{i}^{N_{train}} (\hat{\mathbf{Y}}_i - \mathbf{Y}_i)^T (\hat{\mathbf{Y}}_i - \mathbf{Y}_i) \tag{3.6}$$

Dropout is one popular type of the so-called stochastic regularization technique and has been shown to improve DNN performance in different applications (Srivastava et al. (2014)).

3.1.3.2 Penalized Loss Function

Another popular regularization technique is adding a parameter norm penalty term $\Omega(\boldsymbol{\omega})$ to the loss function $\mathcal{L}_{\mathrm{r}}(\boldsymbol{\omega}; \mathcal{D}_{train})$ to reduce model complexity. The regularized loss function, which we will denote as $\boldsymbol{\ell}_{\mathrm{r}}(\boldsymbol{\omega})$, can be expressed as the following

$$\boldsymbol{\ell}_{\mathrm{r}}(\boldsymbol{\omega}) \propto \mathcal{L}_{\mathrm{r}}(\boldsymbol{\omega}; \mathcal{D}_{train}) + \Omega(\boldsymbol{\omega}) \tag{3.7}$$

Lasso and ridge are among the standard parameters norm regularization choices, and they use L^1 and L^2 penalties, respectively, on the neural networks' parameters. The L^1 regularizer is well known for encouraging sparsity in parameter estimation, while L^2 has the effect of shrinking parameter estimation towards zero (Goodfellow et al. (2016)). Another known regularizer is the elastic net regularizer, a combination of lasso and ridge penalties (Zou and Hastie (2005)).

In our work, we follow an impressive option for the penalty term in the deep learning literature suggested by Gal and Ghahramani (2016), which is an L^2 penalty form (Please see Section 3.3 for more information and why we follow this choice).

3.1.3.3 Other Regularization Techniques

Other regularization methods are typically used to avoid the overfitting issue in the deep learning literature, such as early stopping strategy and data augmentation technique (See Goodfellow et al. (2016), Chapter 7). The early stopping regularizer terminates the optimization if the validation set error does not improve for a specific number of iterations (”Patience”) and returns to the parameters with the lowest validation error. On the other hand, data augmentation avoids overfitting by generating artificial data points in order to increase the size and diversity of the training data points with the goal of improving the deep neural network's generalization to new data.

The mentioned regularization methods are mainly done during the parameters estimation stage, not the prediction stage. The Monto Carlo dropout approach is an exception when applied to get an uncertainties measure for predictions (Refer to Section 3.3 and Gal and Ghahramani (2016) for details). Utilizing one regularizer does not prevent utilizing other regularizers; they can be used together (Goodfellow et al. (2016)).

3.1.4 Activation Functions and Saturation Issue

Earlier sections of this chapter have introduced the concept of hidden layers of the feed neural network. The hidden layers require us to select activation functions to compute the values of the hidden nodes. There are different activation functions available in deep learning literature, such as sigmoid (Eq. 3.8), hyperbolic tangent (Eq. 3.9), and rectified linear unit (Eq. 3.10).

$$g(x) = \frac{1}{1+\exp(-x)} \tag{3.8}$$

$$g(x) = \frac{2}{1+\exp(-2x)} - 1 \tag{3.9}$$

$$g(x) = max(0, x) \tag{3.10}$$

In deep neural networks, the gradient of the loss function tends to vanish when classic activation functions such as sigmoid and hyperbolic tangent are utilized in 3.1 because their bounded range pushes the gradient towards zero except for a limited input region. This phenomenon is called saturation or vanishing gradient, making gradient-based learning very problematic (Goodfellow et al. (2016)). In modern deep learning literature, the rectified linear activation function (ReLU) becomes the default choice for use with the hidden layers of feedforward neural networks because it alleviates the vanishing-gradient problem of the classic activation functions. Another advantage of using ReLU is its capability of imposing a sparsity representation to the neural network by making parts of the hidden layers contain zeros. Therefore, we use the ReLU activation function in this work.

3.2 Parameters Estimation and Optimization

To utilize the prediction power of the neural network, we need to estimate its parameters $\boldsymbol{\omega}$ that best reflect the observation data $\mathcal{D}_{train} = \{\mathbf{X}_i, \mathbf{Y}_i\}_{i=1}^{N_{train}}$. The parameters estimation problem can be cast as a minimization problem of the loss function described in Eq. 3.7. For deep neural networks, the most common algorithm to minimize the loss function is stochastic gradient descent (SGD) and its variants. The SGD uses the objective function's first-order derivative information, the gradient $\nabla_{\boldsymbol{\omega}} \boldsymbol{\ell}_{\mathrm{r}}(\boldsymbol{\omega})$, to iteratively update the neural network's parameters in the direction of the negative gradient of the objective function. The so-called back-propagation algorithm can obtain the gradient of the objective function $\nabla_{\boldsymbol{\omega}} \boldsymbol{\ell}_{\mathrm{r}}(\boldsymbol{\omega})$ with respect to all parameters $\boldsymbol{\omega} = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L+1}$ by taking advantage of the neural network's hierarchical structure through a recursive application of the chain rule (Goodfellow et al. (2016),Chapter 6).

A fundamental and attractive property of SGD is that it introduces stochasticity in the optimization process by replacing the actual deterministic gradient $\nabla_{\boldsymbol{\omega}} \boldsymbol{\ell}_{\mathrm{r}}(\boldsymbol{\omega}; \mathcal{D}_{train})$ with an approximated gradient computed from a randomly sampled small subset, $\mathcal{D}_{mb}$ often called mini-batch, $\nabla_{\boldsymbol{\omega}}\boldsymbol{\ell}_{\text{r}}(\boldsymbol{\omega};\mathcal{D}_{mb})$ of the entire training data points. This property makes the SGD algorithm much faster than the deterministic gradient descent because it updates the values of parameters using only a mini-batch instead of processing the whole training data set to do a single update of the parameters as in the standard deterministic gradient descent. Moreover, SGD can help with prediction performance because of the stochasticity it adds to the learning process (Goodfellow et al. (2016),Ch.8).

While various SGD algorithms are available, we apply the adaptive moment optimization algorithms (ADAM), a widely used variant of the SGD method (Kingma and Ba (2014)). The ADAM algorithm adapts the step size for each neural network parameter by estimating the first and second moments of gradients. The ADAM updating procedure is as follows:

Algorithm 1 The updating rule of ADAM

```
Require: $\alpha$: Stepsize
Require: $0 \leq \beta_1, \beta_2 < 1$: Exponential decay rates
Require: $\boldsymbol{\ell}_{\text{r}}(\boldsymbol{\omega})$: Objective function with parameters $\boldsymbol{\omega}$
Require: Initialization;
  $\boldsymbol{\omega}_0$: Initial parameter vector
  $\mathbf{M}_0 \leftarrow 0$: Initialize the first moment vector
  $\mathbf{V}_0 \leftarrow 0$: Initialize the second moment vector
  $t \leftarrow 0$: Initialize time-step
  while $\boldsymbol{\omega}_t$ not converged do
      $t \leftarrow t + 1$
      $\mathbf{g}_t \leftarrow \nabla_{\boldsymbol{\omega}}\boldsymbol{\ell}_{\text{r}}(\boldsymbol{\omega_{t-1}})$: computing gradients of objective function at time-step $t$
      $\mathbf{M}_t \leftarrow \beta_1\mathbf{M}_{t-1} + (1-\beta_1)g_t$: Update the first moment estimate of the gradients
      $\mathbf{V}_t \leftarrow \beta_2\mathbf{V}_{t-1} + (1-\beta_2){g_t}^2$: Update the second moment estimate of the gradients
      $\hat{\mathbf{M}}_t \leftarrow \frac{\mathbf{M}_t}{(1-\beta_1^t)}$ : Bias correction of first moment estimate
      $\hat{\mathbf{V}}_t \leftarrow \frac{\mathbf{V}_t}{(1-\beta_2^t)}$ : Bias correction of second moment estimate
      $\boldsymbol{\omega}_t \leftarrow \boldsymbol{\omega}_{t-1} - \alpha\frac{\hat{\mathbf{M}}_t}{(\sqrt{\hat{\mathbf{V}}_t+\epsilon})}$: (Update parameters; $\epsilon$ is a small constant to prevent 0 denominator)
  end while
  return $\boldsymbol{\omega}_t$
```

In the applications in Chapter 5, we use the default values for $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\alpha = 0.001$ as proposed by (Kingma and Ba (2014)), and we set $\epsilon = 10^{-7}$, which is the default value in `TensorFlow` package from `R`.

3.3 Uncertainty Quantification in Deep Neural Networks

It is highly desirable to report the uncertainties associated with predictions in the computer emulation literature. We employ the Monto Carlo dropout method to quantify the uncertainties in the neural network predictions (Gal and Ghahramani (2016)). Monto Carlo dropout, or MC-dropout, is a method based on variational Bayes approximation and has become a standard way to quantify uncertainty in deep neural networks.

MC-dropout required us to incorporate the dropout regularizer and the L2 penalty form given by Eq. 3.11 during the training process.

$$\Omega(\boldsymbol{\omega}) = \sum_{l=1}^{L} \left(\frac{P_{keep}}{2\tau N_{train}} ||\mathbf{W}^{(l)}||_2^2 + \frac{1}{2\tau N_{train}} ||\mathbf{b}^{(l)}||_2^2 \right) \tag{3.11}$$

Where $\tau > 0$ is a hyperparameter that controls the overall contribution of the penalty term; $||\mathbf{W}^{(l)}||_2$ denotes the element-wise L^2 norm for $\mathbf{W}^l$, which is given below:

$$||\mathbf{W}^{(l)}||_2 = \sqrt{\sum_{i=1}^{d_l} \sum_{j=1}^{d_{l-1}} \omega_{(l)ij}^2}$$

where $\omega_{(l)ij}$ is the ij^{th} element in $\mathbf{W}^l$; $||\mathbf{b}^{(l)}||_2$ denotes the element-wise L^2 norm for $||\mathbf{b}^{(l)}||$ defined as below:

$$||\mathbf{b}^{(l)}||_2 = \sqrt{\sum_{i=1}^{d_l} b_{(l)i}^2}$$

with $b_{(l)i}$ representing the ij^{th} element in $||\mathbf{b}^{(l)}||$.

Gal and Ghahramani (2016) proved that when incorporating this penalty choice with the dropout, the trained neural networks are equivalent to variational Bayes approximation to a deep Gaussian process with a corresponding structure to our neural network. As a result, one can construct interval predictions for deep neural networks using the Monte Carlo (MC) dropout approach.

The implementation of MC-dropout is straightforward, and the following steps describe how to get the predictive mean $\mathbb{E}(\mathbf{Y})$ and variance $Var(\mathbf{Y})$ for a given input $\mathbf{X}$:

Step 1: Train the neural network with dropout and the L^2 regularization form shown in 3.11. In other words, we minimize the loss function given in 3.7.

Step 2: Apply dropout multiple times (say $\mathbb{T}$ times) during the test time to generate a sample from the predictive distribution of the neural network. In other words, sample $\mathbb{T}$ sets of vectors of realizations from the Bernoulli distribution $\{\mathbf{r}_t^{(1)}, \mathbf{r}_t^{(2)}, \ldots, \mathbf{r}_t^{(l)}, \ldots, \mathbf{r}_t^{(L)}\}_{t=1}^{\mathbb{T}}$ and use them in 3.5, given that we have estimated the neural network's parameters, to perform $\mathbb{T}$ stochastic forward passes through the learned neural network and collect the $\{\hat{\mathbf{Y}}^{(t)}\}_{t=1}^{\mathbb{T}}$ realizations of $\mathbf{Y}$. As a result, we can get a sample for $\mathbf{Y}$, say $\{\hat{\mathbf{Y}}^{(1)}, \hat{\mathbf{Y}}^{(2)}, ..., \hat{\mathbf{Y}}^{(t)}, ..., \hat{\mathbf{Y}}^{(\mathbb{T})}\}$. This Monte Carlo sample is referred to as MC-dropout.

Step 3: Using the generated sample from the predictive distribution $\{\hat{\mathbf{Y}}^{(t)}\}_{t=1}^{\mathbb{T}}$, we can estimate the predictive mean and variance using the following equations:

$$\mathbb{E}(\mathbf{Y}) \approx \frac{1}{\mathbb{T}} \sum_{t=1}^{\mathbb{T}} \hat{\mathbf{Y}}^{(t)} \tag{3.12}$$

$$\begin{aligned} Var(\mathbf{Y}) \approx & (\frac{P_{keep}}{2\tau N_{train}})^{-1}\, \mathbf{I}_n + \\ & \frac{1}{\mathbb{T}} \sum_{t=1}^{\mathbb{T}} \hat{\mathbf{Y}}^{(t)} (\hat{\mathbf{Y}}^{(t)})^T - \mathbb{E}(\mathbf{Y})(\mathbb{E}(\mathbf{Y}))^T \end{aligned} \tag{3.13}$$

where $\mathbf{I}_n$ is the identity matrix of size n and $(\frac{P_{keep}}{2\tau N_{train}})^{-1}$ is the inverse model precision. We see from 3.12 and 3.13 that the predictive mean equals the sample mean of the MC-dropout, and the predictive variance is the inverse model precision plus the sample variance of the MC-dropout. For more information, see Gal and Ghahramani (2016).

Chapter 4

Various-Neighbor Neural Network Emulators via Feature Engineering and Data Augmentation

This chapter introduces a flexible emulator that approximates computer models with high dimensional outputs and limited runs based on deep neural networks. Before describing our various-neighbor neural network emulator, we begin this chapter by describing the data of computer models considered here, reviewing related works, and illustrating the dimensionality problem of training neural networks with high dimensional outputs under the restriction of limited runs. Also, we present a method to incorporate neighboring information into the learning process. Moreover, we propose a data augmentation technique that enables us to include the information of data points with various distances in the learning process to sharpen the performance of the constructed emulator. In the end, we name the deep neural network that exploits the information in the computer model's parameter and spatial space, and that also uses the proposed data augmentation by the various-neighbor neural network (**V3N**) emulator.

4.1 Data Description

4.1.1 Motivation Problems

The Atlantic meridional overturning circulation (AMOC) is a global-scale circulation of ocean water that transports cold salty water from the North Pole towards the equator, and then the heat of the equator warms up ocean water, which is then transported northwards into the North Atlantic. In other words, changes in the density of ocean water cause this circulation. The AMOC is a vital component of the Earth's climate system and plays an essential role in distributing heat and carbon around our planet. In response to human actions, such as burning fossil fuels, the AMOC may continue to weaken. Models have projected that a decline in the AMOC significantly impacts the Earth's climate system (Alley et al. (2007); Chang et al. (2014)).

Currently, we are in a warming climate phase as actual observations indicate an increase in the average ocean water temperature, an increase in melting in ice sheets and glaciers, and a rise in sea level. As projected by computer models, an increase in the earth's average temperature will change the rate of circulation in the AMOC, which will influence the global climate pattern (Alley et al. (2007)).

4.1.2 The University of Victoria Earth System Climate Model (UVic ESCM)

We use the University of Victoria's Earth System Climate Model (UVic ESCM) (Weaver et al. (2001)), which projects the mean ocean temperature from 1955 to 2006. The Uvic ESCM outputs are 3-dimensional spatial fields on a 77 (latitude) $\times$ 100 (longitude) $\times$ 13 (depth) grid. The outputs have been aggregated over longitudes; then, the resulting outputs are on a 77 (latitude) $\times$ 13 (depth) grid. It is customary to aggregate 3-dimensional climate model outputs into 1-dimensional or 2-dimensional to prevent computational problems or because the skill of the models at higher resolution may not always be reliable (Chang et al. (2014); Bhat et al. (2010); Sham Bhat et al. (2012); Schmittner et al. (2009)).

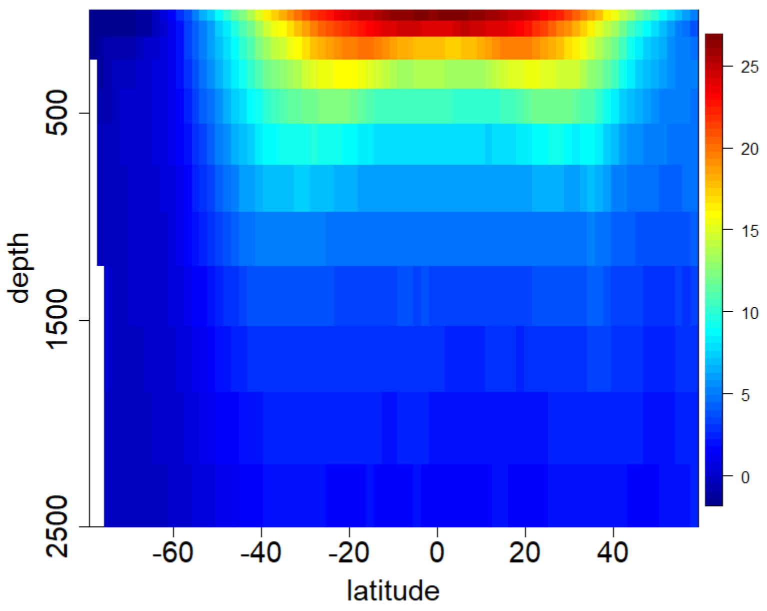


Figure 4.1: *An example of the* 250 *UVic ESCM model runs*

The three parameters that govern the model outputs are vertical ocean background diffusivity (K_{bg}), anthropogenic aerosol scaling factor (A_{scl}), and longwave radiation feedback factor (L_{wr}). Please see Chang et al. (2014) for additional information about the model input parameters. The UVic ESCM is sampled at 250 parameter settings obtained by a factorial design. For more information about the design points and the ensemble outputs, refer to (Sriver et al. (2012)). **Figure 4.1** shows an instance of the 250 model outputs.

4.1.3 The PSU3D-ICE Model

In addition to the UVic ESCM model, we also consider the PSU3D-ICE model (Pollard and DeConto (2009); Chang et al. (2016b)). The PSU3D-ICE model can accurately simulate the long-term behavior of the west antarctic ice sheet. The outputs of the PSU3D-ICE model are spatial fields representing ice thickness patterns on a large grid 171×171, resulting in 29241 spatial locations in the model output (Chang et al. (2016b)).

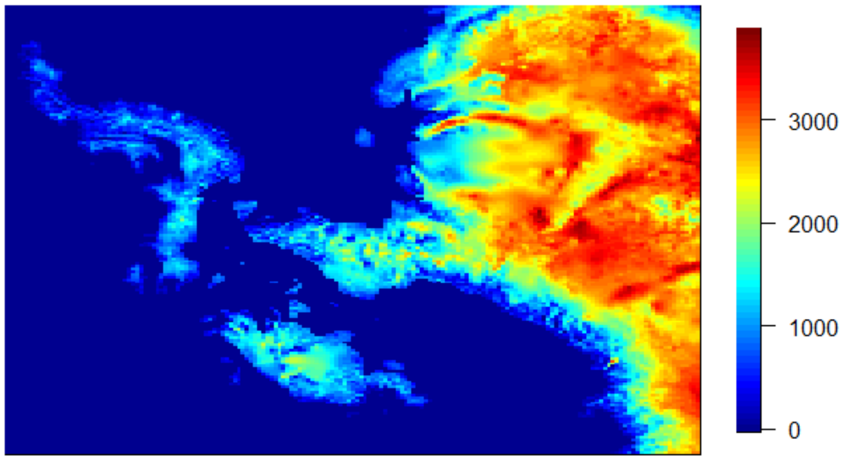


Figure 4.2: *An example of the* 625 *PSU3D-ICE model runs*

The four input parameters that primely control the model outputs are the sub-ice-shelf oceanic melt factor (OCFAC), the calving factor (CALV), the basal sliding coefficient (CRH), and the asthenospheric relaxation folding time (TAU). Please refer to Chang et al. (2016b,a) for a more thorough description of these input parameters. The PSU3D-ICE model outputs are sampled at 625 different parameter settings given by a factorial design, where each input parameter has five different values (Chang et al. (2016b)). A PSU3D-ICE model run is plotted in **figure 4.2**.

4.2 Related Work and Challenges with Large-Scale Applications in Computer Models using Neural Networks

Leveraging neighboring information in data to improve predictive performance is sensible. Wang et al. (2019) proposed a spatial prediction method for a scalar output called Nearest-Neighbor Neural Network for Geostatistics (4N). For better predictive performance, the Nearest-Neighbor Neural Network (4N) offers a new way to incorporate nearby information using the standard multilayer feedforward neural networks (MLP). The 4N method makes no assumptions about the input data format to include neighboring information, as opposed to methods based on a convolutional neural network (CNN) or long short-term memory (LSTM). Typically, CNN is suitable for tasks with grid cell inputs such as images, while LSTM is ideal for sequence input data such as time series (LeCun et al. (1995); Huang et al. (2015); Goodfellow et al. (2016)).

To include neighboring information in the neural network, Wang et al. (2019) offered different ways to engineer input features to the neural network based on neighboring information. For example, Wang et al. (2019) introduced a non-parametric set of features, which consists of the spatial location $\boldsymbol{s}_j$=(latitude, longitude) of the scalar response $Y(\boldsymbol{s}_j)$, the m nearest neighbors to $Y(\boldsymbol{s}_j)$, the m differences in latitudes,and the m differences in longitudes. The non-parametric feature engineering approach (Wang et al. (2019)) would result in $p = 2 + 3m$ input features to the neural network for a scalar output.

This non-parametric feature engineering can be generalized for n-dimensional outputs, resulting in $2 + (m \times n) + 2m$ input features. In the UVic ESCM, the dimension of the outputs is $n = 1001$, and the dimension of the input parameters $\boldsymbol{\theta_i} \in \mathbb{R}^3$ is 3; hence the number of input features will be $p = 3 + (m \times n) + 3m$. This naive generalization of the proposed method in (Wang et al. (2019)) for high dimensional outputs will lead to a massive number of input features, leading to large size of the input layer of the neural network, hence increasing the number of parameters in the neural network. Also, this generalization will lead us to choose the neural network output layer of size n since we have n-dimensional outputs $\mathbf{Y_i} \in \mathbb{R}^n$. Under the constraint of the limited number of runs, a neural network with large input and output layers, which significantly increases the number of neural network parameters, usually makes the training infeasible. The issue even becomes worse for our second application, the PSU3D-ICE model, where the dimension of the output is $n = 29241$.

In the presence of high dimensional outputs and a limited number of runs, it is crucial to engineer the input features appropriately to enable us to choose a suitable architecture of neural networks and make the training feasible. To make the neural network training feasible, we introduce parallel features engineering approach in Section 4.3.1.

The notion of finding nearby information in the input parameter space is unclear. The input parameters have different units and scales. The standard metrics, such as the Euclidean distance, would not appropriately address the anisotropy nature of the computer model data. Also, computer models usually represent very complex systems; large deep neural networks are required to approximate them sufficiently. However, large deep neural networks are susceptible to over-fitting (Goodfellow et al. (2016); Srivastava et al. (2014)), especially in computer model settings where only limited numbers of runs are available. Imposing strong regularization methods, such as weight decay and dropout, can help avoid over-fitting (Goodfellow et al. (2016); Srivastava et al. (2014)). However, such regularization methods can reduce the flexibility of the fitted model and cause over-smoothing. Typically, over-smoothing is not desired in deterministic computer models since we want to interpolate nearly (Ranjan et al. (2011); Gramacy (2020)). To overcome these challenges, we adopt a data augmentation technique into our method in Section 4.3.2.

4.3 Methodology

Let $Y_{ij} = Y(\boldsymbol{\theta}_i, \mathbf{s}_j)$ denotes the observation at the spatial location $\mathbf{s}_j$ and the model parameter setting $\boldsymbol{\theta}_i$. Let $\mathbf{X}_{ij} = (X_{ij1}, ..., X_{ijp})^T$ represent the engineered p features at the spatial location $\mathbf{s}_j$ in the parameter space $\boldsymbol{\Theta}$ using the parallel features engineering approach (more details are in Section 4.3.1). Our approach to construct an emulator consists of the following steps:

(1) Consider the following model:

$$Y_{ij} = f(\mathbf{X}_{ij}) + \epsilon_{ij} \begin{cases} i = 1, 2, \dots, r \\ j = 1, 2, \dots, n \end{cases} \tag{1}$$

where $f(.)$ is a feedforward deep neural network (DNN) and ϵ_{ij} is Gaussian noise.

(2) We engineer the features $\mathbf{X}_{ij}$ for the observation Y_{ij} using the parallel features engineering approach.

(3) To significantly improve the learning process and predictive performance, we incorporate a data augmentation strategy into our approach.

4.3.1 Parallel Features Engineering

The proposed parallel features engineering allows us to include nearby information with high dimensional outputs data without blowing up the number of parameters of the neural networks. One feature of the computer model outputs is that the outputs are observed at the same n known spatial locations. It is standard for computer models to have this feature, but we have highlighted it here because it is a key for the approach in this work. We can consider these n locations as indexes of components in the n-dimensional output $\mathbf{Y_i} = (Y(\boldsymbol{\theta}_i, \mathbf{s}_1), Y(\boldsymbol{\theta}_i, \mathbf{s}_2), ..., Y(\boldsymbol{\theta}_i, \mathbf{s}_j), ..., Y(\boldsymbol{\theta}_i, \mathbf{s}_n))^T$, for example, $\mathbf{s}_j$ is an index of the observation $Y(\boldsymbol{\theta}_i, \mathbf{s}_j)$. At each spatial location $\mathbf{s}_j$, we engineer features for the observation Y_{ij} in the parameter space $\boldsymbol{\Theta}$. In short, we look at a specific spatial location and then extract neighboring information in the computer parameter space. We call this approach parallel features engineering since we can engineer features independently at each spatial location.

Definition 1. ***Nearest Neighborhood*** N_{ij}***:***

At a specific location s_j and a parameter setting $\boldsymbol{\theta_i}$, we define the nearest neighborhood N_{ij} of size m for an observation $Y_{ij} = Y(\boldsymbol{\theta}_i, \boldsymbol{s}_j)$ as the set that has the related information about Y_{ij} itself and the m nearest observations $\{Y_{ij}^1, Y_{ij}^2, ..., Y_{ij}^m\}$ to the Y_{ij} in the parameter space Θ using Euclidean metric.

We consider the following sets of features in N_{ij} to predict Y_{ij}:

- **Basic features** $\{\boldsymbol{\theta}_i, \mathbf{s}_j\}$ **of the query point** Y_{ij}:

  We call the input parameter setting $\boldsymbol{\theta}_i$ and the spatial location $\mathbf{s}_j$ of an observation Y_{ij} by the basic features. The basic features do not depend on the metric we use to decide the nearby observations in parameter space or the size of the neighborhood. Including the basic features as inputs can capture non-stationarity behavior. However, using just the basic features, we do not expect the neural network to completely address the dependence structure in the computer model data.

- **Nearby information in parameter space $\{Y_{ij}^k, \boldsymbol{\theta_i^k}, \boldsymbol{\Delta\theta_i^k}\}$ to Y_{ij}:**

  We engineer different features from N_{ij} for an observation Y_{ij} such as (1) m nearest neighbors $Y_{ij}^k, k = 1, 2, .., m$ in the parameter space $\mathbf{\Theta}$ at the same spatial location $\mathbf{s}_j$, but different parameter settings (2) the parameter settings $\boldsymbol{\theta_i^k}$, $k = 1, 2, .., m$, of the m nearest neighbors, (3) and the differences between the parameter setting of the query observation and the m nearest neighbors $\boldsymbol{\Delta\theta_i^k}, k = 1, 2, .., m$. Including neighboring information in $\mathbf{\Theta}$ as input features makes it possible for neural networks to recognize the underlying dependence structure between the computer model observations at nearby parameter settings. Furthermore, using distance differences $\boldsymbol{\Delta\theta_i^k} = (\theta_{i1} - \theta_{i1}^k, ..., \theta_{iv} - \theta_{iv}^k)^T$ as input features allows the neural network to implicitly learn proper weights in each direction, allowing it to acquire a distance-based prediction emulator.

- **Nearby information in spatial space $\{Y_{ij}^{kc}, \mathbf{s}_j^{kc}\}$ to Y_{ij}^k:**

  In addition to neighboring information in $\mathbf{\Theta}$, we also include neighboring information in $\mathcal{S}$ because of spatial dependence within computer model outputs. In other words, observations at nearby spatial coordinates in $\mathbf{Y_i}$ are highly correlated. We consider C nearest neighbors Y_{ij}^{kc}, $c = 1, 2, .., C$ to the spatial coordinate $\mathbf{s}_j$ of the m nearest neighbors $Y_{ij}^k, k = 1, 2, .., m$ in $\mathbf{\Theta}$. For example, the C nearest neighbors in $\mathcal{S}$ to the spatial coordinate $\mathbf{s}_j$ of first neighbor Y_{ij}^1 in $\mathbf{\Theta}$ are Y_{ij}^{1c}, $c = 1, 2, ..., C$. Also, we consider the spatial coordinates of the C nearest neighbors $Y_{ij}^{kc}, c = 1, 2, .., C$ in $\mathcal{S}$, i.e. we use $\mathbf{s}_j^{kc}$, $c = 1, 2, .., C$, as input features. We anticipate that incorporating neighboring information in spatial space $\mathcal{S}$ can improve the prediction performance because neural networks can benefit from the spatial dependence at nearby spatial locations within the computer model outputs.

Figure 4.3 provides an abstract sketch of the proposed parallel feature engineering approach. Our parallel feature engineering will significantly reduce the number of parameters in the neural network since it leads to an output layer of a single node and a manageable size of the input layer. Therefore, our features engineering will enable us to include neighboring information while simultaneously making the training of neural networks viable under the constraint of high dimensional outputs and a limited number of runs.

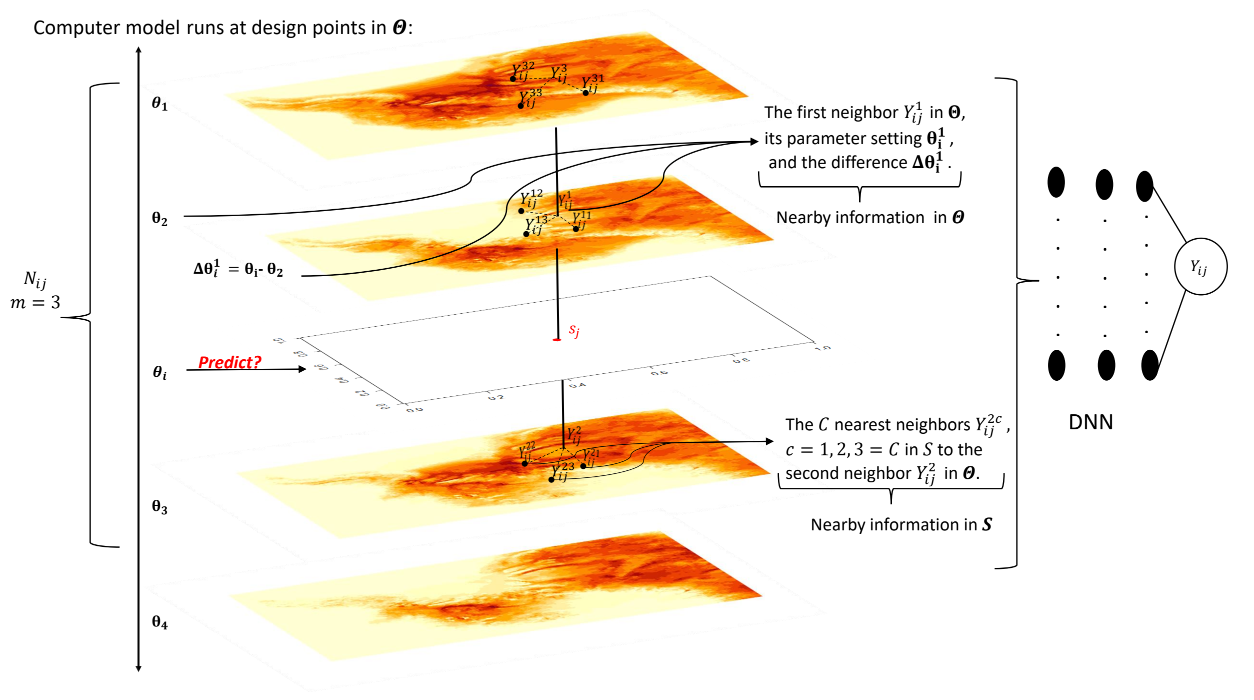

Computer model runs at design points in $\boldsymbol{\Theta}$:
$\boldsymbol{\theta}_1$
$\boldsymbol{\theta}_2$
$\boldsymbol{\theta}_i$
$\boldsymbol{\theta}_3$
$\boldsymbol{\theta}_4$
N_{ij}
$m = 3$
$\Delta\boldsymbol{\theta}_i^1 = \boldsymbol{\theta}_i - \boldsymbol{\theta}_2$
Predict?
s_j
Y_{ij}^{32} Y_{ij}^{3} Y_{ij}^{31} Y_{ij}^{33}
Y_{ij}^{12} Y_{ij}^{1} Y_{ij}^{13} Y_{ij}^{11}
Y_{ij}^{22} Y_{ij}^{2} Y_{ij}^{21} Y_{ij}^{23}
The first neighbor Y_{ij}^1 in $\boldsymbol{\Theta}$, its parameter setting $\boldsymbol{\theta}_i^1$, and the difference $\Delta\boldsymbol{\theta}_i^1$.
Nearby information in $\boldsymbol{\Theta}$
The C nearest neighbors Y_{ij}^{2c}, $c = 1, 2, 3 = C$ in S to the second neighbor Y_{ij}^2 in $\boldsymbol{\Theta}$.
Nearby information in $\boldsymbol{S}$
DNN
Y_{ij}

Figure 4.3: *Illustration of the parallel features engineering approach. Here, $m = C = 3$. To make a prediction at an untried parameter setting $\boldsymbol{\theta_i}$ and at spatial location $\boldsymbol{s}_j$, neighboring information in $\boldsymbol{\Theta}$ and $\mathcal{S}$ can be used through input features to the neural network.*

We consider the following three models based on different sets of input features to the neural network for predicting Y_{ij}:

Model 1:

This model uses the basic features $\{\boldsymbol{\theta}_i,\, \mathbf{s}_j\}$ as input features to predict Y_{ij}. The number of input features is $v + d$.

Model 2:

This model is an extension of **Model 1** and uses the basic features and the nearby information in parameter space. As a result, the number of input features is $v+d+m(1+2v)$. In particular, from N_{ij} of size m, **Model 2** uses the following input features $\{\underbrace{\boldsymbol{\theta}_i, \mathbf{s}_j}_{\substack{\text{basic}\\\text{features}}},\ \underbrace{Y_{ij}^1, \boldsymbol{\theta_i^1}, \boldsymbol{\Delta\theta_i^1}}_{\substack{1^{st}\text{ neighbor}\\\text{information in }\boldsymbol{\Theta}}}, Y_{ij}^2, \boldsymbol{\theta_i^2}, \boldsymbol{\Delta\theta_i^2}, ..., \underbrace{Y_{ij}^m, \boldsymbol{\theta_i^m}, \boldsymbol{\Delta\theta_i^m}}_{\substack{m^{th}\text{ neighbor}\\\text{information in }\boldsymbol{\Theta}}}\}$ to predict Y_{ij}.

Model 3:

We call the model that uses the basic features, nearby information in parameter space, and nearby information in the spatial space **Model 3**, which is an extension of **Model 2** and **Model 1**. This model has $v + d + m(1 + 2v) + mC(1 + d)$ input features and uses the following input features $\{\underbrace{\boldsymbol{\theta}_i, \mathbf{s}_j}_{\substack{\text{basic}\\\text{features}}}, \underbrace{Y_{ij}^1, \boldsymbol{\theta_i^1}, Y_{ij}^{11}, \mathbf{s}_j^{11}, Y_{ij}^{12}, \mathbf{s}_j^{12}, ..., Y_{ij}^{1C}, \mathbf{s}_j^{1C}, \boldsymbol{\Delta\theta_i^1}}_{\substack{1^{st}\text{ neighbor}\\\text{information in }\boldsymbol{\Theta}\text{ and }\mathcal{S}}},\ Y_{ij}^2, \boldsymbol{\theta_i^2}, Y_{ij}^{21}, \mathbf{s}_j^{21}, Y_{ij}^{22}, \mathbf{s}_j^{22}, ..., Y_{ij}^{2C}, \mathbf{s}_j^{2C}$

$, \boldsymbol{\Delta\theta_i^2}, ..., \underbrace{Y_{ij}^m, \boldsymbol{\theta_i^m}, Y_{ij}^{m1}, \mathbf{s}_j^{m1}, Y_{ij}^{m2}, \mathbf{s}_j^{m2}, ..., Y_{ij}^{mC}, \mathbf{s}_j^{mC}, \boldsymbol{\Delta\theta_i^m}}_{\substack{m^{th}\text{ neighbor}\\\text{information in }\boldsymbol{\Theta}\text{ and }\mathcal{S}}}\}$ to predict Y_{ij}.

Based on the choice of input features, we refer to **Model 1** as the **no-neighbor** model, while models **2** and **3** as the **nearest-neighbor** models.

4.3.2 Data Augmentation

4.3.2.1 Motivation

The concept of closeness in the parameter space $\boldsymbol{\Theta}$ is not straightforward. Standard metrics ignore the anisotropy of the input parameter space, or the input parameters are entirely on different scales, which is common in computer experiments. However, to implement the **nearest-neighbor** models, we need to find the m nearest neighbors that are close to the query point in Euclidean or some other metrics. Naturally, this leads us to question if it is optimal to rely only on the nearest neighbors' information throughout the input parameter space in the neural network learning process.

Finding the best data points can be influential for many machine-learning algorithms like nearest-neighbor algorithm-based methods (Hastie and Tibshirani (1995)) or Gaussian process approximation methods (Stein et al. (2004); Gramacy and Apley (2015); Vecchia (1988)). Vecchia (1988) observes that the m nearest neighbors are usually not optimal. (Gramacy and Apley (2015); Gramacy (2020)) argues that distance data points can be of great value, particularly when there are strong correlations, because their information provides long-range dependence information. Stein et al. (2004) shows that parameter estimation can be improved considerably by including distance data points; however, this is not necessarily beneficial when the goal is prediction.

In machine-learning literature, the performance of many algorithms can be remarkably improved by utilizing good data points rather than naively using the nearest neighbors throughout the input space in terms of Euclidean. Consequently, this raises the question of how can we do better than using the closest neighbors in our proposed method?

4.3.2.2 Data Points of Various Distances

Motivated by the belief that we can do better than only relying on the m nearest neighbors points in terms of Euclidean, we adopt a data augmentation into our method to enhance the learning ability of neural networks and eventually improve the prediction performance. Our data augmentation technique increases the size and diversity of training data, and as a result, we will have more training data with diverse ranges of information. Therefore, our model will experience many scenarios during the training stage and see different cases. Hence, improving the learning dynamic of the neural network.

The idea is that we will consider multiple neighborhoods with different information structures for each observation. Then we will train the neural network to predict the same observation from the multiple neighborhoods with varying information structures. By doing our data augmentation, we will have training data with various distances. Thus, our model can learn how to handle data points of various distances. **Figure 4.4** depicts the basic notion of the proposed data augmentation strategy. To be explicit in applying the data augmentation, we define what we mean by the multiple neighborhoods and describe how to utilize them to create more training data points.

Definition 2. ***Neighborhood*** $N_{ij}^{(u)}$***:***
At a specific location s_j *and a parameter setting* $\boldsymbol{\theta_i}$*, we define the neighborhood* $N_{ij}^{(u)}$ *of size* m *for an observation* Y_{ij} *as the set that has the related information about* Y_{ij} *itself and the* m *nearest observations starting from the* u *nearby observation* $\{Y_{ij}^{u}, Y_{ij}^{u+1}, ..., Y_{ij}^{u+m-1}\}$ *to the* Y_{ij} *in the parameter space* Θ *using Euclidean distance.*

Technically, we can consider $r - m + 1$ neighborhoods of size m for each data point. That is, u can take the following values $1, 2, 3, ..., r - m, r - m + 1$. If we consider only one neighborhood with

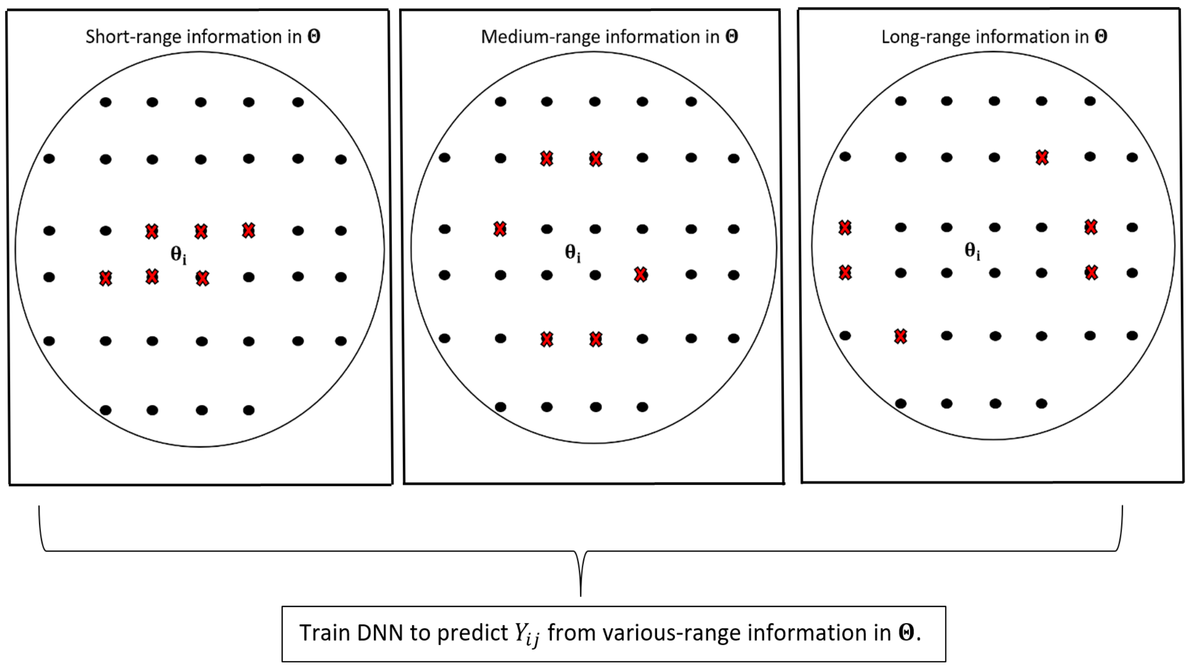

Short-range information in $\boldsymbol{\Theta}$
Medium-range information in $\boldsymbol{\Theta}$
Long-range information in $\boldsymbol{\Theta}$
θ_i
θ_i
θ_i
Train DNN to predict Y_{ij} from various-range information in $\boldsymbol{\Theta}$.

Figure 4.4: *Sketch of the proposed data augmentation.* ***Black dots*** *denotes design points in* $\boldsymbol{\Theta}$. ***Red cross marks*** *represent the selected design points form which we engineered input features for DNN to predict* Y_{ij}. *The* ***left*** *panel uses the information on the short-distance points. The* ***middle*** *panel uses the information on the medium-distance points. The* ***right*** *panel uses the information on the long-distance points.*

$u = 1$ for each observation, i.e. $N_{ij}^{(1)}$, we are just utilizing the m nearest neighbors in the learning stage. However, the purpose of our data augmentation is to increase the size and diversity of the training data to encourage the neural network to accurately learn a robust representation of the engineered features through the use of various-range information.

4.3.2.3 Data Augmentation Details

To apply the data augmentation, we need to consider multiple neighborhoods in $\boldsymbol{\Theta}$, each with different information structures. For example, if we consider the following neighborhoods $N_{ij}^{(1)}$, $N_{ij}^{(m+1)}$, and $N_{ij}^{(2m+1)}$ for each observation Y_{ij} in **Model 2**, our model will be trained to predict each observation Y_{ij} from the information that corresponds to neighborhoods $N_{ij}^{(1)}$, $N_{ij}^{(m+1)}$, and

$N_{ij}^{(2m+1)}$. So, we will train our neural network to predict the same Y_{ij} from the following input features:

- From $N_{ij}^{(1)}$:

$$\{ \underbrace{\boldsymbol{\theta}_i, \mathbf{s}_j}_{\text{basic features}}, \underbrace{Y_{ij}^1, \boldsymbol{\theta_i^1}, \boldsymbol{\Delta\theta_i^1}, Y_{ij}^2, \boldsymbol{\theta_i^2}, \boldsymbol{\Delta\theta_i^2}, ..., Y_{ij}^m, \boldsymbol{\theta_i^m}, \boldsymbol{\Delta\theta_i^m}}_{\text{the features in } N_{ij}^{(1)}} \}$$

  In our applications, we set the size of the neighborhood to equal six. $N_{ij}^{(1)}$ is visually given in the left panel of **figure 4.4** where $m = 6$.

- From $N_{ij}^{(m+1)}$:

$$\{ \underbrace{\boldsymbol{\theta}_i, \mathbf{s}_j}_{\text{basic features}}, \underbrace{Y_{ij}^{m+1}, \boldsymbol{\theta_i^{m+1}}, \boldsymbol{\Delta\theta_i^{m+1}}, Y_{ij}^{m+2}, \boldsymbol{\theta_i^{m+2}}, \boldsymbol{\Delta\theta_i^{m+2}}, ..., Y_{ij}^{2m}, \boldsymbol{\theta_i^{2m}}, \boldsymbol{\Delta\theta_i^{2m}}}_{\text{the features in } N_{ij}^{(2m+1)}} \}$$

  The middle panel of **figure 4.4** sketches $N_{ij}^{(m+1)}$ for $m = 6$.

- From $N_{ij}^{(2m+1)}$:

$$\{ \underbrace{\boldsymbol{\theta}_i, \mathbf{s}_j}_{\text{basic features}}, \underbrace{Y_{ij}^{2m+1}, \boldsymbol{\theta_i^{2m+1}}, \boldsymbol{\Delta\theta_i^{2m+1}}, Y_{ij}^{2m+2}, \boldsymbol{\theta_i^{2m+2}}, \boldsymbol{\Delta\theta_i^{2m+2}}, ..., Y_{ij}^{3m}, \boldsymbol{\theta_i^{3m}}, \boldsymbol{\Delta\theta_i^{3m}}}_{\text{the features in } N_{ij}^{(2m+1)}} \}$$

  The right panel of **figure 4.4** shows $N_{ij}^{(2m+1)}$ where $m = 6$.

The proposed data augmentation will incorporate dependence information of diverse ranges in neural networks' parameters learning process (parameter estimation) instead of only using short-range dependence information. Consequently, neural networks will be encouraged to learn more robust representations of the underlying computer models. Also, the way our data augmentation introduces more training points will encourage the neural network to understand the structure of the input parameter space $\boldsymbol{\Theta}$. By considering multiple neighborhoods in $\boldsymbol{\Theta}$ for the same Y_{ij}, the neural network will see how the neighboring information varies differently in $\boldsymbol{\Theta}$ and learn meaningful weights for the differences in each direction $\boldsymbol{\Delta\theta_i^k} = (\theta_{i1} - \theta_1^k, ..., \theta_{iv} - \theta_v^k)^T$.

Deep-large neural network structures are essential for developing an interpolator nearly of deterministic computer models, but high-capacity structures are prone to overfitting with limited runs. Reducing the flexibility of neural network structures, either by considering smaller structures or imposing strong regularization techniques, such as weight decay and dropout, can discourage the emulator from interpolating, which is not desired for deterministic computer models. Our data augmentation enables us to use deep-large neural network structures required to fit highly complex responses because it can act as a regularizer and help avoid overfitting.

When models **2** and **3** incorporate our data augmentation in the training process, we refer to them by **various-neighbor** models. Also, let us denote **Model 2** with data augmentation by **Model 2+Aug** and **Model 3** with data augmentation by **Model 3+Aug**.

Chapter 5

Applications

5.1 UVic ESCM Model Application

In this section, we investigate the prediction accuracy of the proposed method using Uvic ESCM data. We design two test scenarios to show that using only the nearest neighbors in the learning process of neural networks can lead to an emulator with a poor generalization and a highly sensitive emulator to the underlying test scenarios.

Through two test cases, we show that it is necessary for the neural network to incorporate various-range information using the proposed data augmentation during the training process in order to learn a robust representation and greatly improve the emulator's prediction accuracy.

In the end, we compare our method with the parallel partial Gaussian process emulator (Gu and Berger (2016)), an existing state-of-the-art emulation method for high dimensional outputs. The parallel partial Gaussian process emulator, **PPGP**, is a statistical method to emulate computer models that run on massive grids. **PPGP**, also called robust Gaussian stochastic process emulation, was developed by `Mengyang Gu` as part of his Ph.D. dissertation (Gu (2016)) and has shown promising results on large-scale computer models emulation problems (Gu and Berger (2016)); therefore, we have decided to compare it with our method. The **PPGP** is implemented with `R` package `RobustGaSP` (Gu et al. (2018)).

5.1.1 Investigating Data Augmentation's Effectiveness

5.1.1.1 Designing Test Studies

To carefully examine the proposed method, we design two test cases using the data from the Uvic ESCM Model. Recall for the Uvic ESCM Model; the model outputs $\mathbf{Y_i}$ are obtained at $r = 250$ design points $\boldsymbol{\theta_1}, ..., \boldsymbol{\theta_i}, ..., \boldsymbol{\theta_{250}}$, specified by a factorial design where the dimension of input space is $v = 3$, $\boldsymbol{\theta_i} \in \boldsymbol{\Theta} \subseteq \boldsymbol{R}^3$. We have three input parameters $\boldsymbol{\theta} = (\theta_1, \theta_2, \theta_3)$ with five levels for θ_1, five levels for θ_2, ten levels for θ_3. **Figure 5.1** shows the design points in its space given by the factorial design.

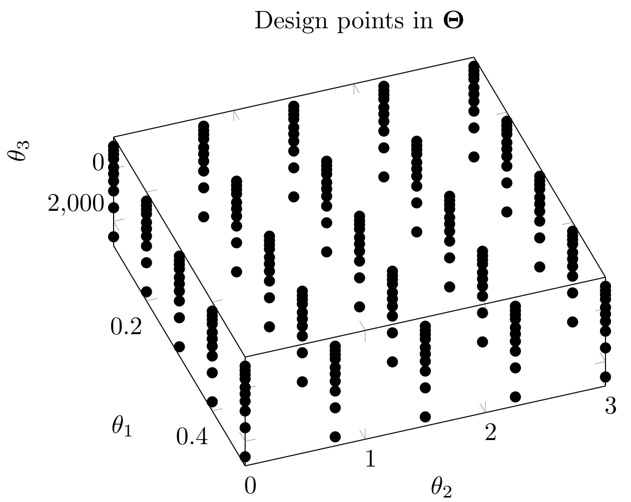


Figure 5.1: *The Uvic ESCM Model's design points.*

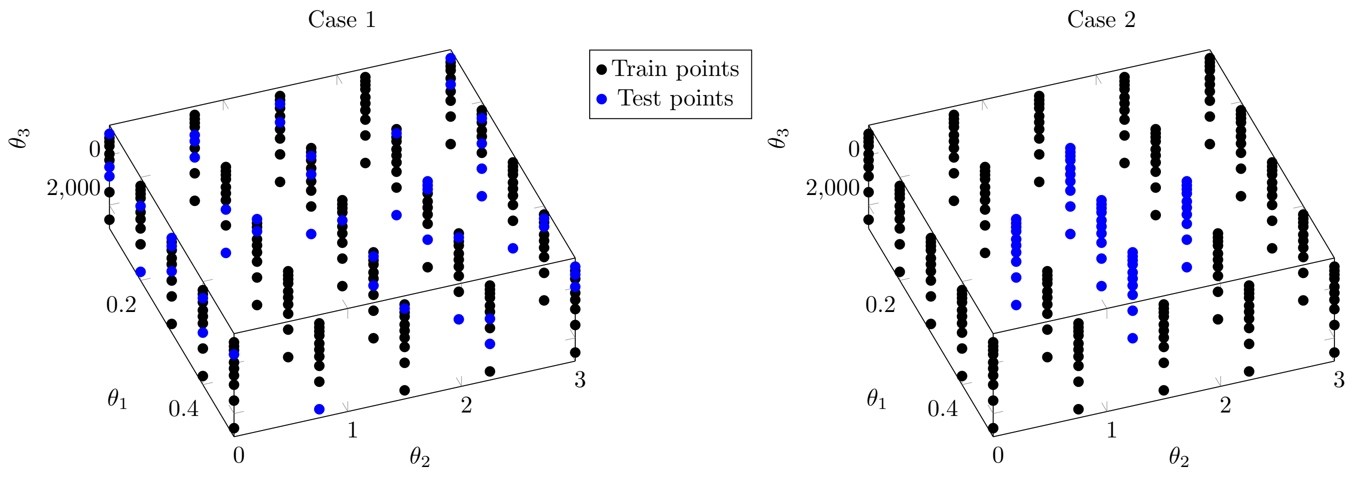


Figure 5.2: *Visualization of the two test situations.* ***Left:*** *The test design points (blue dots) are chosen randomly in the input space.* ***Right:*** *The test design points (blue dots) are selected to cause a gap in the input space.*

We divided the 250 runs into 50 runs as test design points ($r^* = 50$) and 200 as train design points ($r = 200$). We consider two ways to choose the 50 test points. **Figure 5.2** offers a visual illustration of the two test situations. For the **case one** scenario, we choose the test points randomly. As a result, the train points will stay spread out in Θ, which is advisable in computer models to extract maximum information about the underlying behavior of computer models. For the **case two** scenario, we choose the test points in a way that causes a gap in the middle of the input space. Consequently, the **case two** test will resemble a scenario where an uneven sampling or clumpiness of design points exists, which is not recommended when the prediction is the primary concern. However, emulators are generally applied in higher dimensional (i.e., more than 2d) input space with limited runs. As a result, it is not always clear how to balance the design points in the input space.

5.1.1.2 Implementation Details

We have used `TensorFlow` and `Keras` packages from `R` to implement the **no-neighbor**, **nearest-neighbor**, and **various-neighbor** models. For all models, we take 20% of the training data as a validation data set $\mathcal{D}_{\texttt{val}}$ to tune the models' hyperparameters, such as the size of neighborhoods, number of hidden layers and their sizes, epochs, etc. We apply the dropout technique and an early stopping rule to help avoid overfitting. The 'early stopping' rule terminates the training if the validation error does not decrease for several epochs (the patience).

For **Model 1**, we use five hidden layers, each with 128 units. We set the patience to 30 in the early stopping rule, the dropout rate to 0.00005, the epoch to 150, and the mini-batch size to 512.

For the **nearest-neighbor** and **various-neighbor** models, we set the size of the neighborhood to 6 in $\mathbf{\Theta}$ as well as in $\mathcal{S}$ spaces to do the proposed features engineering in Section 4.3.1.

For the **nearest-neighbor** models, We use five layers with hidden units ranging from 128 to 256. We use mini-batches of size 256, set the number of epochs to 200, and take $p_{drop} = 0.00001$ (the dropout rate). Also, we use the early stopping with patience =30.

We also have implemented the **various-neighbor** models with five hidden layers, each with 128 units. We utilize mini-batches of size 512 and let the number of epochs to 300. We set the patience to 100 and let $p_{drop} = 0.00005$. To have wildly diverse training data, we have applied our data augmentation using fourteen neighborhoods ,$N_{ij}^{(u)}$, with different information structures. To be specific, u takes the following values $1, 10, 20, ..., 120, 130$ in **definition 5.3**. As a result, the training data set will grow fourteen times its initial size.

5.1.1.3 Comparing the Performance based on Neighboring Choice

We use the root-mean-square prediction error (**RMSPE**), whose formula is provided below, to assess the predictive performance of the presented models.

$$\mathbf{RMSPE} = \sqrt{\frac{\sum_{j=1}^{n}\sum_{i=1}^{r^*}(Y_{ij}^* - \hat{Y}_{ij}^*)^2}{nr^*}},$$

where $\hat{Y}_{ij}^*$ is the predicted values of the Y_{ij}^* in the test data. We also plot the boxplots of the root-mean-square prediction errors for the examined models.

1) Case one:

We have implemented the **no-neighbor**, **nearest-neighbor**, and **various-neighbor** models, and **RMSPE** is computed along the way and reported in **table 5.1**. Moreover, boxplots comparing the root-mean-square prediction errors are shown in **Figure 5.3**.

Figure 5.3 and **table 5.1** show that, generally, **Model 1** has the poorest predictive performance, and incorporating information of other outputs in the learning process of neural networks, whether nearest distance information or various distance information, leads to a better version of the constructed emulator. By comparing **Model 2** with **Model 3**, we see that including information in the spatial domain $\mathcal{S}$ typically leads to slight improvement. Also, we see that the proposed data augmentation has drastically improved the predictive performance of **Model 2** and **Model 3**. Furthermore, we can see that data augmentation's effect is much larger than the effect of including nearby information in spatial space, but it is still useful. The predictive performance of **Model 2+Aug** and **Model 3+Aug** is very competitive, but overall, **Model 3+Aug** performs slightly better.

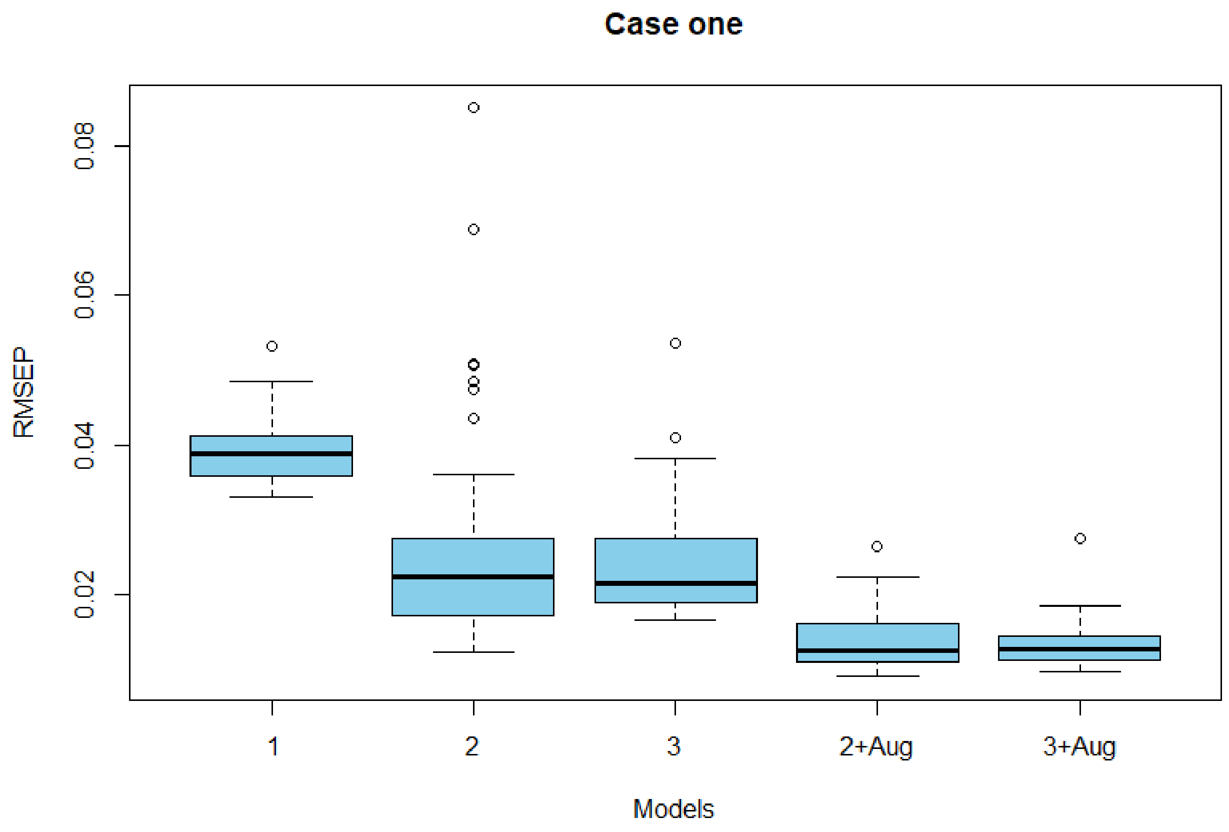


Figure 5.3: *Comparison of root mean-squared prediction errors for **Model 1**, **Model 2**, **Model 3**, **Model 2+Aug**, **Model 3+Aug** under case one scenario.*

2) Case two:

Under this test scenario, we also have implemented the **no-neighbor**, **nearest-neighbor**, and **various-neighbor** models. Then the **RMSPE** is calculated to compare the predictive performance and reported in **table 5.2**. **Figure 5.4** shows boxplots comparing the root-mean-squared errors under the **case two** test.

Figure 5.4 and **table 5.2** show that the predictive performance of **Model 2** and **Model 3** is not good as in the **case one** test. **Model 2** has the poorest predictive performance in terms of

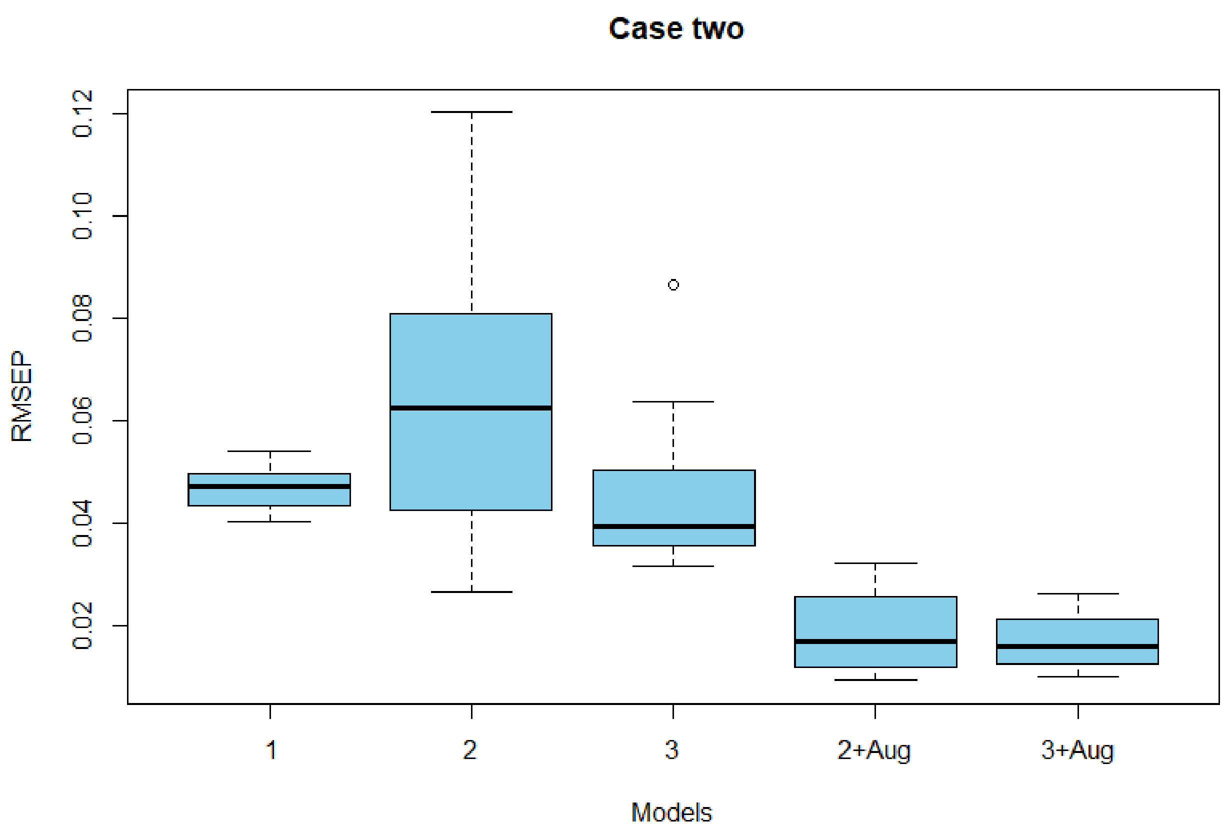


Figure 5.4: *Comparison of the root mean-squared prediction errors for **Model 1**, **Model 2**, **Model 3**, **Model 2+Aug**, **Model 3+Aug** under case two scenario.*

RMSPE. By looking at **Model 2** and **Model 3**, we see it is worthwhile to include the information in the spatial domain. Similar to the **case one** test, the predictive performance of **Model 2** and **Model 3** has significantly improved after utilizing the data augmentation in the training process. This observation reveals the protection our data augmentation can offer against challenging scenarios such as a gap in the input space, an uneven sampling, or sparsity in data. Due to the powerful effect of our data augmentation, the prediction accuracy of **Model 2+Aug** and **Model 3+Aug** becomes very similar; however, it is not surprising that **Model 3+Aug** performs slightly better than **Model 2+Aug** because it incorporates information in both spaces Θ and $\mathcal{S}$.

5.1.1.4 Necessity of Incorporating our Data Augmentation in Training Process

From **figure 5.3** and **figure 5.4**, as well as **table 5.1** and **table 5.2**, we see that training with data augmentation has remarkably improved the prediction performance of the **nearest-neighbor** models in both test case scenarios. The remarkable improvement highlights the necessity of utilizing our data augmentation in the training process to learn a better and more robust representation of the underlying structure of computer model data. The need to employ the proposed data augmentation in the training process of the **nearest-neighbor** models can be explained from various perspectives.

case one scenario	**without** augmentation	**with** augmentation
Model 1	0.03956	not available
Model 2	0.02978	0.01425
Model 3	0.02532	**0.01356**

Table 5.1: ***RMSPE*** *under case one test.*

case two scenario	**without** augmentation	**with** augmentation
Model 1	0.04712	not available
Model 2	0.06773	0.01941
Model 3	0.04571	**0.0175**

Table 5.2: ***RMSPE*** *under case two test.*

An explanation is that our data augmentation increases the size and diversity of training points, making the training data more comprehensive and therefore encouraging the stochastic gradient descent to find a set of weights that works for different case scenarios. Therefore, our data augmentation prevents neural networks from overfitting simple-to-learn input features and ignoring difficult-to-learn input features. For illustration, due to the highly correlated data of computer models, it is possible that the neural network can learn to predict query points primarily based on the first nearest neighbor Y_{ij}^1 and mostly ignore the other input features such as differences

$\boldsymbol{\Delta\theta_i^k}, k = 1, 2, .., m$ in $\boldsymbol{\Theta}$. As a result, Y_{ij}^1 will dominate the learning process (the parameters update in the stochastic gradient descent); hence, the trained model would be susceptible to Y_{ij}^1.

Another possible explanation is that the proposed data augmentation allows us to include multiple range dependence information for the same observation, thus enabling the neural network to understand how the correlation between the observations changes in the input space. Consequently, the trained neural network learns proper weights in each difference between the input parameters, $\boldsymbol{\Delta\theta_i^k} = (\theta_{i1} - \theta_{i1}^k, ..., \theta_{iv} - \theta_{iv}^k)^T$, and eventually acknowledges the anisotropic behavior in data from computer models.

Overall, **Model 3+Aug** shows the best prediction performance regardless of the underlying test scenario. Furthermore, **Model 3+Aug** takes advantage of the information in both spaces. Therefore, we use **Model 3+Aug** to build an emulator and compare it with standard state-of-art emulation methods for large outputs. For the rest of this work, we call **Model 3+Aug** by 'Various-Neighbor Neural Network emulator' or simply **V3N** to emphasize the proposed data augmentation's role in the learning process.

5.1.2 Comparison between our V3N and PPGP

5.1.2.1 Test Scenarios and Evaluation Criteria

For both **V3N** and **PPGP**, the prediction performance is evaluated under the same two test scenarios shown in **Figure 5.2**. We use $r = 200$ runs to construct the emulator and $r^* = 50$ runs to evaluate the predictive performance in both test scenarios.

To evaluate the predictive performance, we use different criteria: **RMSPE** given in 5.1.1.3, the 95% empirical coverage of the prediction intervals ($\mathbf{P_{CI}}$), and the average length of the 95% prediction intervals ($\mathbf{L_{CI}}$). A better emulator will have a smaller **RMSPE**, empirical coverages close to the nominal confidence level, and shorter average prediction interval lengths. The formulas of the $\mathbf{P_{CI}}$ and $\mathbf{L_{CI}}$ criteria are given below:

$$\boldsymbol{P_{CI}} = \frac{1}{nr^*} \sum_{j=1}^{n} \sum_{i=1}^{r^*} 1\{Y_{ij}^* \in P_{CI_{ij}}\},$$

$$\boldsymbol{L_{CI}} = \frac{1}{nr^*} \sum_{j=1}^{n} \sum_{i=1}^{r^*} L_{CI_{ij}},$$

where $\hat{Y}_{ij}^*$ is the predicted values of the true Y_{ij}^* in the test data; $P_{CI_{ij}}$ and $L_{CI_{ij}}$ are the empirical coverage and the length of the 95% prediction interval.

To do uncertainty quantification in our **V3N**, we use Monto Carlo dropout technique (Gal and Ghahramani (2016)).

5.1.2.2 Implementation Details

We have utilized `TensorFlow` and `Keras` packages from `R` in implementing the **V3N** emulator. For the **V3N** method, we have applied the proposed data augmentation with the same fourteen neighborhood structures specified in Section 5.1.1.2. Consequently, the size of the training data set $\mathcal{D}_{\texttt{train}}$ will grow fourteen times the original size. Furthermore, we randomly keep 20% of $\mathcal{D}_{\texttt{train}}$ as a validation data set $\mathcal{D}_{\texttt{val}}$ to tune the deep neural networks hyperparameters.

To apply the proposed features engineering in Section 4.3.1, we first need to choose the size of the neighborhood in both spatial and parameter spaces. We tried values from 15 to 5, and based on the validation set error, we choose the size of the neighborhood to be 6 in both spaces ($m = C = 6$). Also, we use the validation set to tune the number of hidden layers and their size (hidden unites), mini-batch size, and the number of epochs. We use 5 hidden layers, each with 128 units. For the hidden layers, we use the Relu activation function, and for the output layer, we use linear activation function. We use mini-batches of size 512 and set the number of epochs to 300. We use an 'early-stopping' rule, with the rule's patience equal to 100 iterations.

For MC-dropout hyperparameters, we find $\tau = 3000000$ (length-scale) and $p_{drop} = 0.00005$ (dropout probability) results in small validation error and desired coverages of the interval estimates after systemic tuning using the validation set. We use the Adaptive Moments (ADAM) optimization algorithm to fit the neural networks.

5.1.2.3 Comparison Results

1) Case one:

Test one scenario	**RMSPE**	(95%)$\boldsymbol{P_{CI}}$	(95%)$\boldsymbol{L_{CI}}$
V3N	0.0136	0.9763	0.0687
PPGP	0.0086	0.9047	0.0221

Table 5.3: *A comparison of the performance of our emulator and PPGP in terms of various metrics. Both emulators are evaluated based on* $r^* = 50$ *randomly chosen test input parameter settings.*

Table 5.3 shows the results of comparing our method with **PPGP** in terms of the three metrics under the **case one** scenario which was shown in **figure 5.2**. The **PPGP** emulator based on randomly selected test data outperforms our **V3N** emulator in terms of **RMSPE**. However, our method performed very well and has a small RMSPE relative to the range of the predicted output values (-1.68 to 28.47). Similarly, the average length of the prediction interval for the predicted observations is reasonably short over its range. These results indicate that our method can precisely emulate the underlying computer model. In Section S1.1 in the Supplementary Material, we plot some predicted outputs from our method and **PPGP** and compare them with the true outputs under this test scenario.

1) Case two:

Test two scenario	**RMSPE**	$(95\%)\boldsymbol{P_{CI}}$	$(95\%)\boldsymbol{L_{CI}}$
V3N	0.0175	0.9461	0.0699
PPGP	0.0196	0.8139	0.0253

Table 5.4: *A comparison of the performance of our emulator and PPGP in terms of different criteria. Both emulators are evaluated based on* $r^* = 50$ *test input settings, which were chosen in a way to cause a gap in the input space.*

Table 5.4 shows that when there is a gap in the input region, our emulator better fits the underlying computer models than the **PPGP** emulator in terms of **RMSPE**. Also, our method has good coverage for this case test, whereas the **PPGP** has an under coverage issue. Moreover, unlike **PPGP**, the underlying test scenario had little impact on the performance of our method. These results indicate that our **V3N** emulator can handle various problematic scenarios. Please refer to Section S1.2 of the Supplementary Material for a visual comparison under the second case test scenario between the predicted and actual computer outputs.

5.2 PSU3D-ICE Model Application

In this section, we apply our proposed method using more complex data from the PSU3D-ICE model. The ice sheet model has four input parameters $\boldsymbol{\theta} = (\theta_1, \theta_2, \theta_3, \theta_4)$ and is run at 625 different parameter settings specified by a factorial design, with five levels for each input parameter. The data from the PSU3D-ICE model is much larger than the data from the UVic ESCM model, where the number of spatial locations is $n = 29241$, and the number of runs is $r = 625$, which will emphasize the computational benefit of our method over the standard Gaussian process emulator. Moreover, the ice sheet model data has very complicated structures, which will demonstrate the powerful flexibility of deep neural networks in capturing complex structures.

5.2.1 Design of the test study and evaluation criteria

To examine and validate the performance of our **V3N**, we randomly divide the 625 runs into 200 training runs to build the emulator, and $r^* = 425$ testing runs to evaluate the accuracy. Moreover, we compare our **V3N** with **PPGP** in terms of the three metrics specified before in 5.1.1.3 and 5.1.2.1, i.e., **RMSPE**, (95%)$\boldsymbol{P_{CI}}$ and (95%)$\boldsymbol{L_{CI}}$. Again, we used Monto Carlo dropout technique (Gal and Ghahramani (2016)) to quantify uncertainties in our method.

5.2.2 Implementation Details

We utilize the same fourteen neighborhood structures described in Section 5.1.1.1 for our method to employ our data augmentation technique. In addition, we randomly use 20% of $\mathcal{D}_{\texttt{train}}$ as a validation data set $\mathcal{D}_{\texttt{val}}$ to tune the hyperparameters of our method. We try a range from 5 to 10 for m, and based on $\mathcal{D}_{\texttt{val}}$ error; we use $m = C = 6$ in $\boldsymbol{\Theta}$ and $\mathcal{S}$ to engineer the input features mentioned in Section 4.3.1. We use six hidden layers. The first, second, fifth, and sixth hidden layers have 128 units, while the third and fourth hidden layers have 256 units. We use the Relu activation function for the hidden layers as well as for the output layer to eliminate the possibility of predicting a negative sea-level rise. We use mini-batches of size 8192 and set the number of epochs to 100. We employ an 'early-stopping' rule, with the rule's patience set to 50 iterations. To apply MC-dropout, we set $\tau = 2000$ and $p_{drop} = 0.005$, which led to a smaller validation error and desired coverage for the validation set. The parameters of the deep neural network are estimated using ADAM optimizer algorithm. We use the easy-to-use `TensorFlow` and `Keras` packages from `R` to train our **V3N** emulator. To implement **PPGP** emulator, we use `RobustGaSP` package in `R` (Gu et al. (2018))

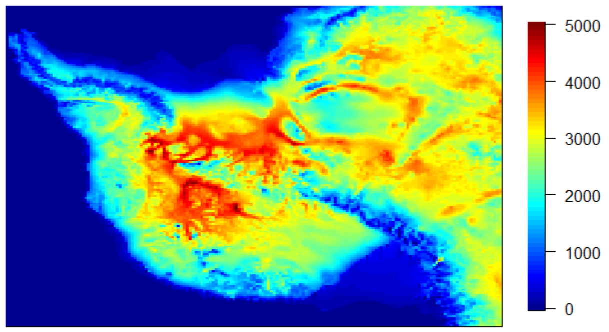
5.2.3 Comparison Results

As mentioned before, we have used $r = 200$ to construct the emulators and we will diagnose the performance based on $r^* = 425$ test input settings. **Table 5.5** shows the comparison results between our method and **PPGP** based on different criteria which they were mentioned in Section 5.1.2.1.

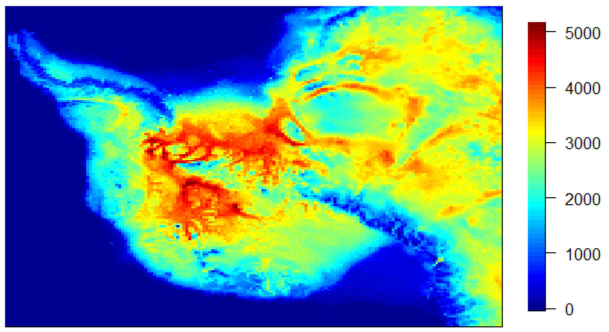
Methods	**RMSPE**	(95%)$\boldsymbol{P_{CI}}$	(95%)$\boldsymbol{L_{CI}}$
V3N	198.175	0.9694	479.624
PPGP	229.912	0.9359	520.301

Table 5.5: *A comparison of the performance of our emulator and PPGP in terms of various metrics. Both emulators are evaluated based on* $r^* = 425$ *randomly chosen test input parameter settings.*

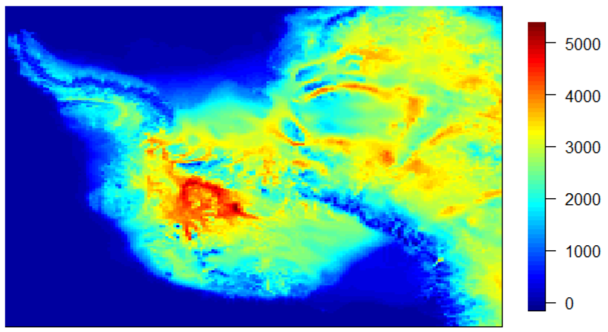
The results in **table 5.5** demonstrate that our method performs better than **PPGP** for the ice sheet model. These results show that the **RMSPE** of **V3N** emulator is lower than the **RMSPE** of **PPGP**. Generally, we believe that both emulators have reasonably small **RMSPE**, especially when considering that the predicted output values range from 0 to 5252.866. Regarding uncertainty quantification, we can see that both emulators produce empirical coverages close to the nominal confidence level 95% ;however, our method has a shorter average length of the 95% prediction intervals. In addition, we provide examples of visual comparisons between the original outputs and the emulated outputs from the considered emulators in **Figures 5.5**, **5.6** and **5.7** to support the results in **table 5.5** and show good predictive performance of our method.



Original model output

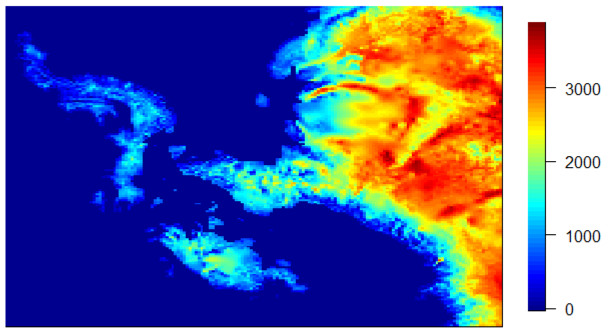


V3N emulated output

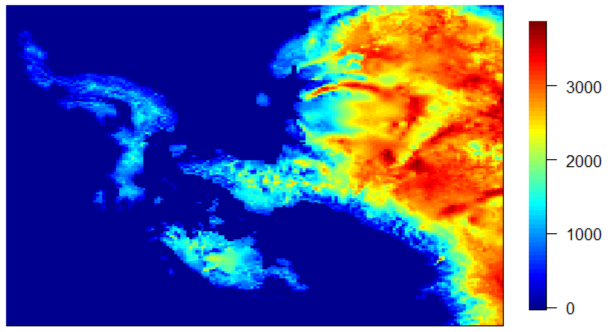


PPGP emulated output

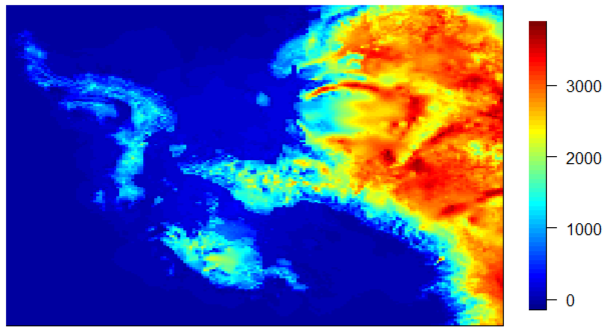
Figure 5.5: *A visual comparison between an original computer output and the emulated outputs from* ***V3N*** *and* ***PPGP****. The top plot shows an output from the test data. The low-left plot shows the predicted output using our method. The low-right plot show the predicted output by* ***PPGP****.*



Actual model run

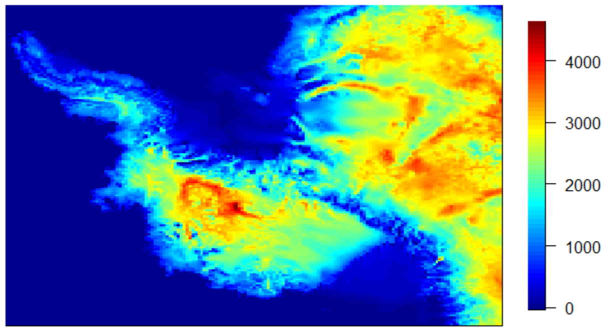


Predicted output via **V3N**

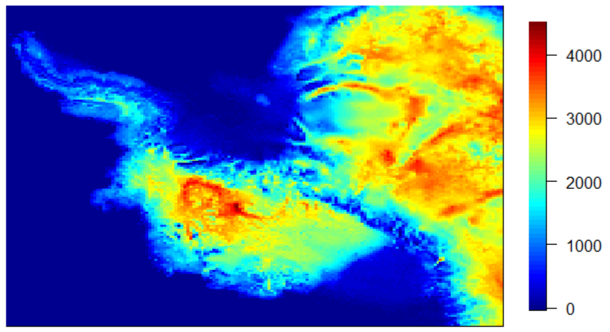


Predicted output via **PPGP**

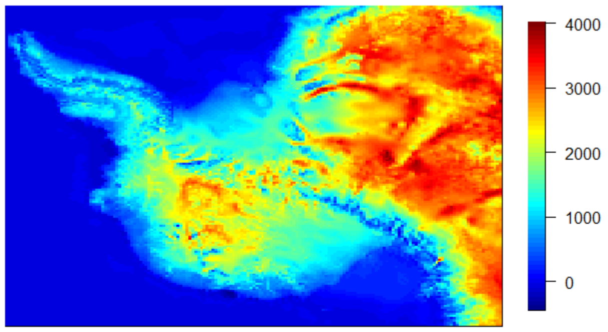
Figure 5.6: *Another visual comparison example between an original computer output and the predicted outputs from **V3N** and **PPGP**. The top plot shows an output from the test data. The low-left plot shows the predicted output using our method. The low-right plot show the predicted output by **PPGP**.*



Real model run



Emulated run via **V3N**



Emulated run via **PPGP**

Figure 5.7: *A visual comparison between original computer and predicted outputs from* ***V3N*** *and* ***PPGP****. The top plot shows an output from the test data.*

Chapter 6

Summary

In this work, we have developed a new methodology for emulating computer models with high dimensional outputs based on deep neural networks. To overcome the dimensionality problem of training neural networks with high dimensional outputs and limited runs, we have proposed a new way to engineer input features that leads to a trainable architecture for neural networks. Our proposed features engineering enables us to incorporate neighboring information between and within computer model outputs in the learning process of neural networks. Furthermore, To enhance the learning ability and improve the predictive performance of neural networks, we have developed a data augmentation technique that enables us to utilize the outputs' information of various ranges in the training stage of the neural networks. Also, we have investigated the effect of our data augmentation on the prediction performance of our method. We have demonstrated the necessity of training with our data augmentation technique to learn a robust representation of the underlying computer model and improve the prediction accuracy for the proposed method, and we have provided multiple explanations for this finding.

Moreover, we have shown that our proposed emulator **V3N** can accurately emulate computer model outputs, and the prediction performance is robust under different scenarios. We have compared our emulator with an existing state-of-art emulator for high dimensional outputs, **PPGP**, using computer data from the UVic ESCM model and the PSU3D-ICE model. Our emulator produces promising results and outperforms **PPGP** under a gap scenario in the input space in the context of UVic ESCM model outputs and outperforms **PPGP** in the context of PSU3D-ICE model outputs. Also, We have demonstrated that our **V3N** can capture complicated structures like those in the ice sheet model while **PPGP** can fail.

Deep neural networks are naturally suited to handle data with complicated structures; however, their performance relies on the amount of training data. In other words, deep neural networks often perform well in big data problems (a large amount of training data). Using deep neural networks to address emulation problems with high dimensional outputs and limited runs can be challenging since the emulation problems are seen as small data problems (limited runs) in the context of deep neural networks. We believe our work will be a valuable contribution since it provides a methodology that successfully applies deep neural networks to approximate computer models with large outputs and limited runs, and it outperforms an existing state-of-art emulation method under various scenarios.

Bibliography

Alley, R. B., Berntsen, T., Bindoff, N., Chidthaisong, A., Friedlingstein, P., Gregory, J., Hegerl, G., Heimann, M., Hewitson, B., Hoskins, B., et al. (2007). Summary for policymakers.

Banerjee, S., Gelfand, A. E., Finley, A. O., and Sang, H. (2008). Gaussian predictive process models for large spatial data sets. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 70(4):825–848.

Bayarri, M., Walsh, D., Berger, J., Cafeo, J., Garcia-Donato, G., Liu, F., Palomo, J., Parthasarathy, R., Paulo, R., and Sacks, J. (2007). Computer model validation with functional output. *The Annals of Statistics*, 35(5):1874–1906.

Bhat, K. S., Haran, M., Goes, M., and Chen, M. (2010). Computer model calibration with multivariate spatial output: A case study. *Frontiers of statistical decision making and Bayesian analysis*, 111:168–184.

Bhatnagar, S., Chang, W., and Wang, S. K. J. (2020). Computer model calibration with time series data using deep learning and quantile regression. *arXiv preprint arXiv:2008.13066*.

Block, H.-D. (1962). The perceptron: A model for brain functioning. i. *Reviews of Modern Physics*, 34(1):123.

Booker, A. J., Dennis, J. E., Frank, P. D., Serafini, D. B., Torczon, V., and Trosset, M. W. (1999). A rigorous framework for optimization of expensive functions by surrogates. *Structural optimization*, 17(1):1–13.

Chang, W., Haran, M., Applegate, P., and Pollard, D. (2016a). Calibrating an ice sheet model using high-dimensional binary spatial data. *Journal of the American Statistical Association*, 111(513):57–72.

Chang, W., Haran, M., Applegate, P., and Pollard, D. (2016b). Improving ice sheet model calibration using paleoclimate and modern data. *The Annals of Applied Statistics*, 10(4):2274–2302.

Chang, W., Haran, M., Olson, R., and Keller, K. (2014). Fast dimension-reduced climate model calibration and the effect of data aggregation. *The Annals of Applied Statistics*, 8(2):649–673.

Chen, M., Jiang, H., Liao, W., and Zhao, T. (2019). Efficient approximation of deep relu networks for functions on low dimensional manifolds. *Advances in neural information processing systems*, 32.

Conti, S. and O'Hagan, A. (2010). Bayesian emulation of complex multi-output and dynamic computer models. *Journal of statistical planning and inference*, 140(3):640–651.

Cressie, N. and Johannesson, G. (2008). Fixed rank kriging for very large spatial data sets. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 70(1):209–226.

Cressie, N. A. (1993). Statistics for spatial data. john willy and sons. *Inc., New York*.

Datta, A., Banerjee, S., Finley, A. O., and Gelfand, A. E. (2016). Hierarchical nearest-neighbor gaussian process models for large geostatistical datasets. *Journal of the American Statistical Association*, 111(514):800–812.

Deng, L., Hinton, G., and Kingsbury, B. (2013). New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8599–8603. IEEE.

Di, Q., Kloog, I., Koutrakis, P., Lyapustin, A., Wang, Y., and Schwartz, J. (2016). Assessing pm2. 5 exposures with high spatiotemporal resolution across the continental united states. *Environmental science & technology*, 50(9):4712–4721.

Finley, A. O., Sang, H., Banerjee, S., and Gelfand, A. E. (2009). Improving the performance of predictive process modeling for large datasets. *Computational statistics & data analysis*, 53(8):2873–2884.

Gal, Y. and Ghahramani, Z. (2016). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

Gramacy, R. B. (2020). *Surrogates: Gaussian process modeling, design, and optimization for the applied sciences*. Chapman and Hall/CRC.

Gramacy, R. B. and Apley, D. W. (2015). Local gaussian process approximation for large computer experiments. *Journal of Computational and Graphical Statistics*, 24(2):561–578.

Gramacy, R. B. and Lee, H. K. (2012). Cases for the nugget in modeling computer experiments. *Statistics and Computing*, 22(3):713–722.

Gramacy, R. B. and Lee, H. K. H. (2008). Bayesian treed gaussian process models with an application to computer modeling. *Journal of the American Statistical Association*, 103(483):1119–1130.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee.

Gu, M. (2016). *Robust uncertainty quantification and scalable computation for computer models with massive output.* PhD thesis, Duke University.

Gu, M. and Berger, J. O. (2016). Parallel partial gaussian process emulation for computer models with massive output. *The Annals of Applied Statistics*, 10(3):1317–1347.

Gu, M., Palomo, J., and Berger, J. O. (2018). Robustgasp: Robust gaussian stochastic process emulation in r. *arXiv preprint arXiv:1801.01874*.

Hastie, T. and Tibshirani, R. (1995). Discriminant adaptive nearest neighbor classification and regression. *Advances in neural information processing systems*, 8.

Higdon, D., Gattiker, J., Williams, B., and Rightley, M. (2008). Computer model calibration using high-dimensional output. *Journal of the American Statistical Association*, 103(482):570–583.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

Huang, Z., Xu, W., and Yu, K. (2015). Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*.

Kaufman, C. G., Schervish, M. J., and Nychka, D. W. (2008). Covariance tapering for likelihood-based estimation in large spatial data sets. *Journal of the American Statistical Association*, 103(484):1545–1555.

Kennedy, M. C. and O'Hagan, A. (2001). Bayesian calibration of computer models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(3):425–464.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

LeCun, Y., Bengio, Y., et al. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995.

Liang, S. and Srikant, R. (2016). Why deep neural networks for function approximation? *arXiv preprint arXiv:1610.04161*.

Neal, R. M. (1997). Monte carlo implementation of gaussian process models for bayesian regression and classification. *arXiv preprint physics/9701026*.

Oakley, J. E. and O'Hagan, A. (2004). Probabilistic sensitivity analysis of complex models: a bayesian approach. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 66(3):751–769.

Pepelyshev, A. (2010). The role of the nugget term in the gaussian process method. In *mODa 9–Advances in Model-Oriented Design and Analysis*, pages 149–156. Springer.

Poggio, T., Mhaskar, H., Rosasco, L., Miranda, B., and Liao, Q. (2017). Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review. *International Journal of Automation and Computing*, 14(5):503–519.

Pollard, D. and DeConto, R. M. (2009). Modelling west antarctic ice sheet growth and collapse through the past five million years. *Nature*, 458(7236):329–332.

Ranjan, P., Haynes, R., and Karsten, R. (2011). A computationally stable approach to gaussian process interpolation of deterministic computer simulation data. *Technometrics*, 53(4):366–378.

Rougier, J. (2008). Efficient emulators for multivariate deterministic functions. *Journal of Computational and Graphical Statistics*, 17(4):827–843.

Sacks, J., Schiller, S. B., and Welch, W. J. (1989). Designs for computer experiments. *Technometrics*, 31(1):41–47.

Santner, T. J., Williams, B. J., Notz, W. I., and Williams, B. J. (2003). *The design and analysis of computer experiments*, volume 1. Springer.

Schmittner, A., Urban, N. M., Keller, K., and Matthews, D. (2009). Using tracer observations to reduce the uncertainty of ocean diapycnal mixing and climate–carbon cycle projections. *Global Biogeochemical Cycles*, 23(4).

Sham Bhat, K., Haran, M., Olson, R., and Keller, K. (2012). Inferring likelihoods and climate system characteristics from climate models and multiple tracers. *Environmetrics*, 23(4):345–362.

Smith, R. C. (2013). *Uncertainty quantification: theory, implementation, and applications*, volume 12. Siam.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

Sriver, R. L., Urban, N. M., Olson, R., and Keller, K. (2012). Toward a physically plausible upper bound of sea-level rise projections. *Climatic Change*, 115(3):893–902.

Stein, M. L., Chi, Z., and Welty, L. J. (2004). Approximating likelihoods for large spatial data sets. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 66(2):275–296.

Tripathy, R. K. and Bilionis, I. (2018). Deep uq: Learning deep neural network surrogate models for high dimensional uncertainty quantification. *Journal of computational physics*, 375:565–588.

Ugarte, M. D. (2015). Banerjee, s. carlin, bp, and gelfand, ae hierarchical modeling and analysis for spatial data. crc press/chapman & hall. monographs on statistics and applied probability 135, boca raton, florida, 2015. 562 pp. 99.95. isbn-13: 978-1-4398-1917-3 (hardcover).

Vecchia, A. V. (1988). Estimation and model identification for continuous spatial processes. *Journal of the Royal Statistical Society: Series B (Methodological)*, 50(2):297–312.

Wang, H., Guan, Y., and Reich, B. (2019). Nearest-neighbor neural networks for geostatistics. In *2019 International Conference on Data Mining Workshops (ICDMW)*, pages 196–205. IEEE.

Weaver, A. J., Eby, M., Wiebe, E. C., Bitz, C. M., Duffy, P. B., Ewen, T. L., Fanning, A. F., Holland, M. M., MacFadyen, A., Matthews, H. D., et al. (2001). The uvic earth system climate model: Model description, climatology, and applications to past, present and future climates. *Atmosphere-Ocean*, 39(4):361–428.

Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the royal statistical society: series B (statistical methodology)*, 67(2):301–320.

Chapter 7

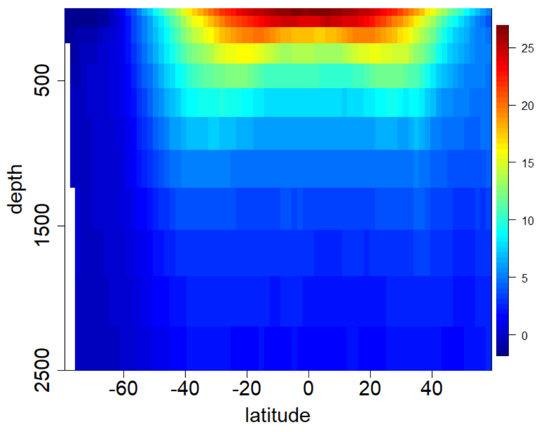
Supplementary Material

This chapter provides Supplement to Chapter 5, where we compare the performance of our **V3N** with **PPGP** under the two test scenarios specified in Section 5.1.1.1.

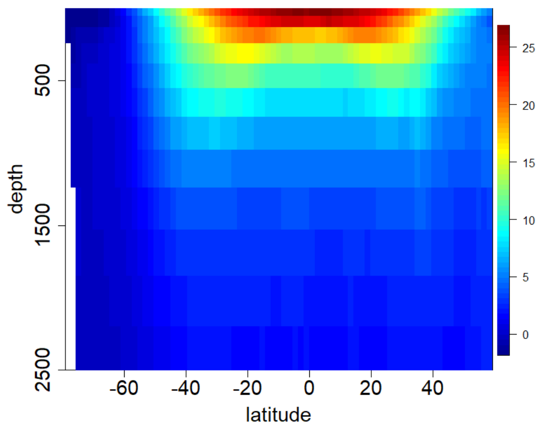
S1 UVic ESCM Model Application

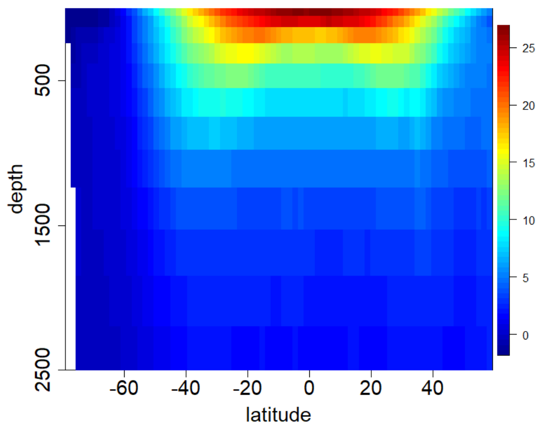
We provide a visual comparison between the emulated outputs from our method and **PPGP** under the two case scenarios for the UVic ESCM model.

S1.1 Visual Comparisons under the First Case Test Scenario



Original model output

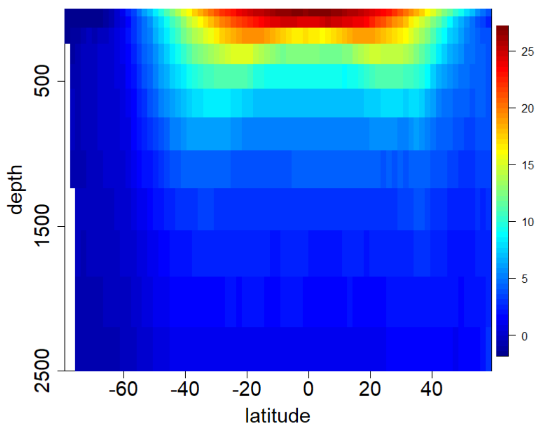




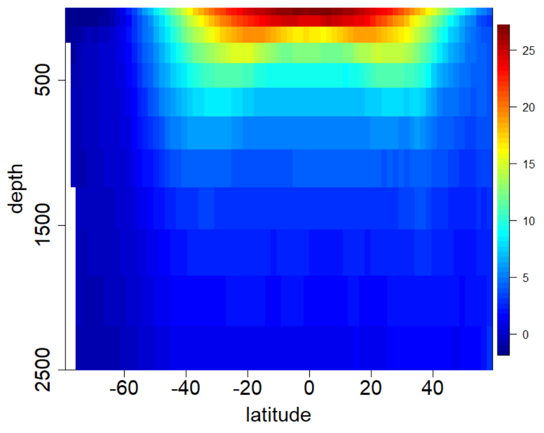
Emulated output via V3N

Emulated output via PPGP

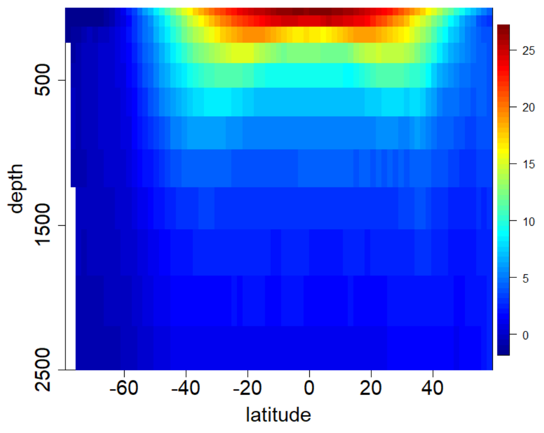
Figure S1: A UVic ESCM run (top plot). The bottom plots show comparison between the emulated run using **V3N** (bottom left plot) and the emulated run using **PPGP** (bottom right plot).



Original model output



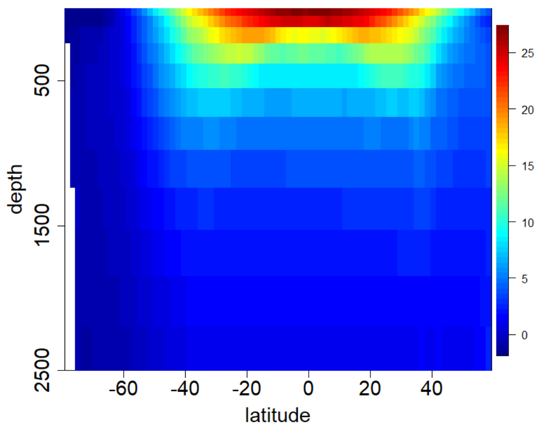
Emulated output via V3N



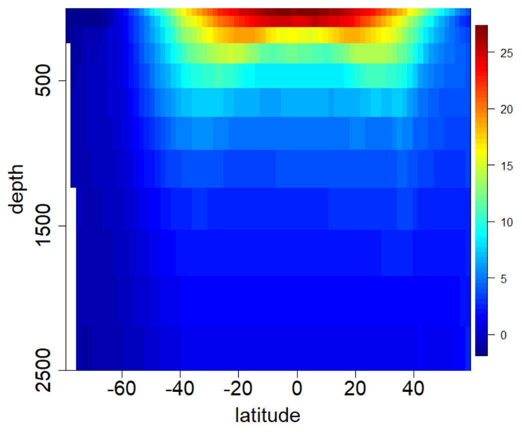
Emulated output via PPGP

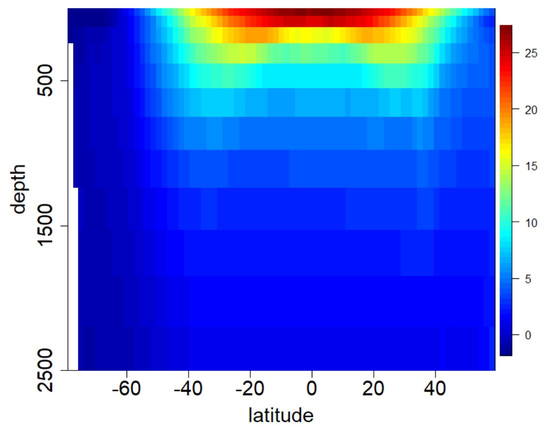
Figure S2: Another visual comparison between the true run (top plot) and the emulated outputs from our **V3N** (bottom left plot) and **PPGP** (bottom right plot) under the first case test scenario.

S1.2 Visual Comparisons under the Second Case Test Scenario



Original model output

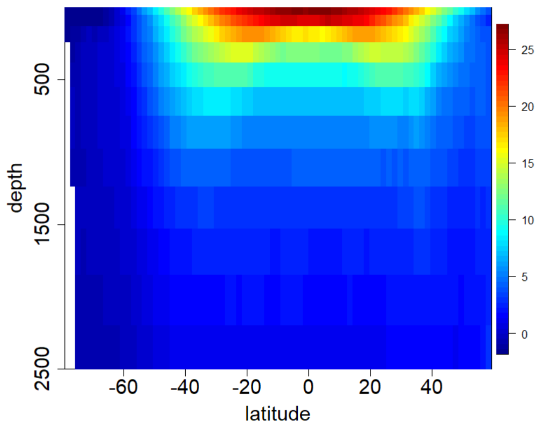




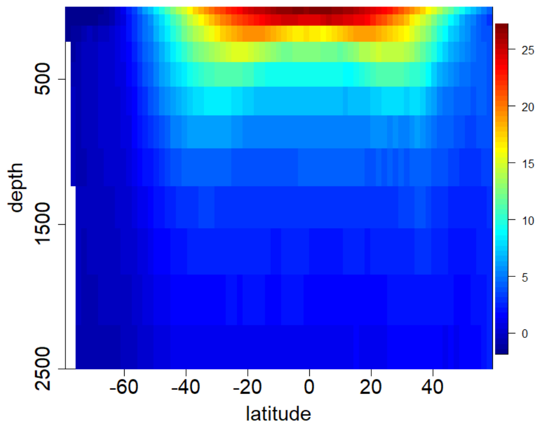
Emulated output via V3N

Emulated output via PPGP

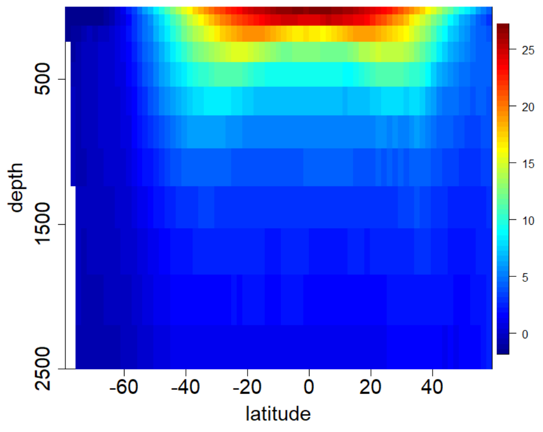
Figure S3: Plots of the actual run (top plot) and the predicted outputs from our **V3N** (bottom left plot) and **PPGP** (bottom right plot) under the second case test.



Original model output



Emulated output via V3N



Emulated output via PPGP

Figure S4: Another visual comparison between the computer model output (top) and the emulated outputs from our method (bottom left) and **PPGP** (bottom right).