PILOT STUDY: MACHINE LEARNING AND DEEP LEARNING STUDY FOR FLUID STRUCTURE INTERACTION PROBLEMS

FINAL PROJECT REPORT

by

Zarak K. Kasi
Barbara G. Simpson
Michael H. Scott

Oregon State University

Sponsorship
PacTrans

for

Pacific Northwest Transportation Consortium (PacTrans)
USDOT University Transportation Center for Federal Region 10
University of Washington
More Hall 112, Box 352700
Seattle, WA 98195-2700

In cooperation with U.S. Department of Transportation,
Office of the Assistant Secretary for Research and Technology (OST-R)

DISCLAIMER

The contents of this report reflect the views of the authors, who are responsible for the facts and the accuracy of the information presented herein. This document is disseminated under the sponsorship of the U.S. Department of Transportation's University Transportation Centers Program, in the interest of information exchange. The Pacific Northwest Transportation Consortium, the U.S. Government and matching sponsor assume no liability for the contents or use thereof.

TECHNICAL REPORT DOCUMENTATION PAGE

1. Report No.	2. Government Accession No. 01723940	3. Recipient's Catalog No.
4. Title and Subtitle Pilot Study: Machine Learning and Deep Learning study for Fluid Structure Interaction problems		**5. Report Date** 07/29/2022 **6. Performing Organization Code**
7. Author(s) and Affiliations Zarak K. Kasi, Oregon State University Barbara G. Simpson, Oregon State University; 0000-0002-3661-9548 Michael H. Scott, Oregon State University 0000-0001-5898-5090		**8. Performing Organization Report No.** 2019-S-OSU-2
9. Performing Organization Name and Address PacTrans Pacific Northwest Transportation Consortium University Transportation Center for Federal Region 10 University of Washington More Hall 112 Seattle, WA 98195-2700		**10. Work Unit No. (TRAIS)** **11. Contract or Grant No.** 69A3551747110
12. Sponsoring Organization Name and Address United States Department of Transportation Research and Innovative Technology Administration 1200 New Jersey Avenue, SE Washington, DC 20590		**13. Type of Report and Period Covered** Project draft report 08/01/2022 **14. Sponsoring Agency Code**

15. Supplementary Notes

Report uploaded to: www.pactrans.org

16. Abstract

Coastal bridges are critical to emergency response after extreme events and are vulnerable to cascading seismic-tsunami events. After the 2011 earthquake and subsequent tsunami in Japan, instances of damage and collapse were observed in Japanese bridges that survived the earthquake but failed under the hydrodynamic loads induced by the tsunami. The Pacific Northwest in the United States could experience similar tsunami hazards. To ensure reliable mobility after extreme events, it is necessary to understand, model, and design bridge response for tsunami loading. However, studies on wave-structure interaction are constrained by the financial cost of experiments and the computational cost of computational fluid dynamics (CFD) and fluid-structure interaction (FSI) simulations. To practically perform such simulations with reduced computational cost, a pilot study that used machine learning algorithms for basic structural engineering problems is presented. Similar machine learning models can eventually be used to estimate the tsunami loading on bridges based on structural properties and flow conditions. Machine learning (ML) and deep learning (DL) algorithms, when trained for a specific problem, can produce faster results than finite element methods (FEM). Nonetheless, ML and DL algorithms are data-driven and could produce unreliable results when evaluated outside the training data domain. The interpretability of ML and DL algorithms can also be lost during the training of the model.

The reliability and interpretability of ML and DL can be resolved by introducing physics into the ML and DL architectures. This project studied the performance of data-driven and physics-informed DL algorithms in structural engineering applications. The DL algorithm was studied for static and dynamic problems using single-degree-of-freedom (SDOF) oscillators representing a simplified model of a bridge pier. Physics was introduced into the DL algorithm by extracting the residual from the finite-element analysis framework, OpenSees, and integrating it with the loss function during training. The performance of the DL algorithm with and without physics was evaluated by using different loss functions, activation functions, and other hyperparameters. For an SDOF for linear static and linear dynamic problems, the data-driven and physics-informed deep learning algorithms produced similar results. Moreover, if an appropriate neural network architecture was utilized, the DL models were able to extrapolate well beyond the test data. Although the studied cases were for relatively simple SDOF linear static and dynamic problems, DL algorithms have the potential to produce reliable results for multi-degree-of-freedom systems, including the relevant physics. The approach of introducing OpenSees along with the ML and DL algorithms also presents an opportunity for engineers to produce fast and reliable results by supplementing analyses with ML and DL techniques. Nonlinear problems, multiple-degrees-of-freedom systems, and FSI studies, including the residual during the learning process, should be assessed to evaluate the performance of DL algorithms beyond the simple structural systems presented herein.

17. Key Words	18. Distribution Statement		
Physics Informed Machine Learning, Deep Learning, Fluid structure Interaction, OpenSees			
19. Security Classification (of this report)	**20. Security Classification (of this page)**	**21. No. of Pages**	**22. Price**
Unclassified.	Unclassified.	60	N/A

SI* (MODERN METRIC) CONVERSION FACTORS

APPROXIMATE CONVERSIONS TO SI UNITS				
Symbol	**When You Know**	**Multiply By**	**To Find**	**Symbol**
		LENGTH		
in	inches	25.4	millimeters	mm
ft	feet	0.305	meters	m
yd	yards	0.914	meters	m
mi	miles	1.61	kilometers	km
		AREA		
in^2	square inches	645.2	square millimeters	mm^2
ft^2	square feet	0.093	square meters	m^2
yd^2	square yard	0.836	square meters	m^2
ac	acres	0.405	hectares	ha
mi^2	square miles	2.59	square kilometers	km^2
		VOLUME		
fl oz	fluid ounces	29.57	milliliters	mL
gal	gallons	3.785	liters	L
ft^3	cubic feet	0.028	cubic meters	m^3
yd^3	cubic yards	0.765	cubic meters	m^3
	NOTE: volumes greater than 1000 L shall be shown in m^3			
		MASS		
oz	ounces	28.35	grams	g
lb	pounds	0.454	kilograms	kg
T	short tons (2000 lb)	0.907	megagrams (or "metric ton")	Mg (or "t")
		TEMPERATURE (exact degrees)		
°F	Fahrenheit	5 (F-32)/9 or (F-32)/1.8	Celsius	°C
		ILLUMINATION		
fc	foot-candles	10.76	lux	lx
fl	foot-Lamberts	3.426	candela/m^2	cd/m^2
		FORCE and PRESSURE or STRESS		
lbf	poundforce	4.45	newtons	N
lbf/in^2	poundforce per square inch	6.89	kilopascals	kPa

APPROXIMATE CONVERSIONS FROM SI UNITS				
Symbol	**When You Know**	**Multiply By**	**To Find**	**Symbol**
		LENGTH		
mm	millimeters	0.039	inches	in
m	meters	3.28	feet	ft
m	meters	1.09	yards	yd
km	kilometers	0.621	miles	mi
		AREA		
mm^2	square millimeters	0.0016	square inches	in^2
m^2	square meters	10.764	square feet	ft^2
m^2	square meters	1.195	square yards	yd^2
ha	hectares	2.47	acres	ac
km^2	square kilometers	0.386	square miles	mi^2
		VOLUME		
mL	milliliters	0.034	fluid ounces	fl oz
L	liters	0.264	gallons	gal
m^3	cubic meters	35.314	cubic feet	ft^3
m^3	cubic meters	1.307	cubic yards	yd^3
		MASS		
g	grams	0.035	ounces	oz
kg	kilograms	2.202	pounds	lb
Mg (or "t")	megagrams (or "metric ton")	1.103	short tons (2000 lb)	T
		TEMPERATURE (exact degrees)		
°C	Celsius	1.8C+32	Fahrenheit	°F
		ILLUMINATION		
lx	lux	0.0929	foot-candles	fc
cd/m^2	candela/m^2	0.2919	foot-Lamberts	fl
		FORCE and PRESSURE or STRESS		
N	newtons	0.225	poundforce	lbf
kPa	kilopascals	0.145	poundforce per square inch	lbf/in^2

*SI is the symbol for the International System of Units. Appropriate rounding should be made to comply with Section 4 of ASTM E380.
(Revised March 2003)

Contents

List of Figures

List of Tables

LIST OF ABBREVIATIONS

Adam	Adaptive moment estimation
AI	Artificial intelligence
ANN	Artificial neural network
CFD	Computational fluid dynamics
CNN	Convolutional neural network
DL	Deep learning
FEM	Finite element methods
FS	Feature scaling
FSI	Fluid-structure interaction
HT	Hyperbolic tangent
LS	Logistic Sigmoid
LSTM	Long short-term memory
MDOF	Multiple degrees of freedom
ML	Machine learning
MLP	Multi-layer perceptron
MSE	Mean squared error
NN	Neural network
OpenSees	Open System for Earthquake Engineering Simulation
PDE	Partial differential equation
PFEM	Particle finite element method
PGNN	Physics guided neural network
PINN	Physics-informed neural network
PS	Pareto scaling
PT	Power transformation
ReLU	Rectified Linear Unit
RNN	Recurrent neural network
SciANN	Scientific computational with artificial neural network
SDOF	Single-degree-of-freedom

TN	Tangent normalization
VSS	Variable stability scaling

ACKNOWLEDGMENTS

The research conducted in this report was funded by a research grant titled, “Pilot Study: Machine Learning and Deep Learning study for Fluid Structure Interaction Problems,” funded by the Pacific Northwest Transportation Consortium (PacTrans). The research was carried out at Oregon State University. Oregon State University provided matching funds for faculty time on the project. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of PacTrans or other participants in the research program.

EXECUTIVE SUMMARY

Coastal bridges are critical to emergency response after extreme events and are vulnerable to cascading seismic-tsunami events. After the 2011 earthquake and subsequent tsunami in Japan, instances of damage and collapse were observed in Japanese bridges that survived the earthquake but failed under the hydrodynamic loads induced by the tsunami. The Pacific Northwest in the United States could experience similar tsunami hazards. To ensure reliable mobility after extreme events, it is necessary to understand, model, and design bridge response for tsunami loading. However, studies on wave-structure interaction are constrained by the financial cost of experiments and the computational cost of computational fluid dynamics (CFD) and fluid-structure interaction (FSI) simulations. To practically perform such simulations with reduced computational cost, a pilot study, which uses machine learning algorithms for basic structural engineering problems, is presented. Similar machine learning models can eventually be used to estimate the tsunami loading on bridges based on structural properties and flow conditions. Machine learning (ML) and deep learning (DL) algorithms, when trained for a specific problem, can produce faster results than finite element methods (FEM). Nonetheless, ML and DL algorithms are data-driven and could produce unreliable results when evaluated outside the training data domain. The interpretability of ML and DL algorithms can also be lost during the training of the model.

The reliability and interpretability of ML and DL can be resolved by introducing physics into the ML and DL architectures. This project studied the performance of data-driven and physics-informed DL algorithms in structural engineering applications. After a brief overview of ML and DL, this report presents a study of the DL algorithm for static and dynamic problems using single-degree-of-freedom (SDOF) oscillators representing a simplified model of a bridge pier. Physics was introduced into the DL algorithm by extracting the residual from the finite-element analysis framework, OpenSees, and integrating it with the loss function during training. The performance of the DL algorithm with and without physics was evaluated by using different loss functions, activation functions, and other hyperparameters. For an SDOF for linear static and linear dynamic problems, the data-driven and physics-informed deep learning algorithms produced similar results. Moreover, if an appropriate neural network architecture was utilized, the DL models were able to extrapolate well beyond the test data.

Although the studied cases were for relatively simple SDOF linear static and dynamic problems, DL algorithms have the potential to produce reliable results for multi-degree-of-freedom systems, including the relevant physics. The approach of introducing OpenSees along with the ML and DL algorithms also presents an opportunity for engineers to have fast and reliable results by supplementing analyses with ML and DL techniques. Nonlinear problems, multiple degrees of freedom systems, and FSI studies, including the residual during the learning process, should be assessed to evaluate the performance of DL algorithms beyond the simple structural systems presented herein.

1. INTRODUCTION

Coastal bridges are vulnerable to tsunami damage. The 2011 Tohoku tsunami in Japan resulted in damage to over 250 bridges, delaying post-event recovery and emergency response. Bridges that survived the earthquake failed under the subsequent tsunami due to the hydrodynamic demands. The Pacific Northwest has similar hazards. Many bridges serve as lifelines for the coastal regions, which are critical for the mobility of people and goods and to provide post-event emergency response. Thus, it is essential to understand the response of coastal bridges under tsunami loading to ensure safety after extreme events. However, the financial cost of experimental tests makes physically simulating the response of structures to tsunami loadings very expensive. On the other hand, simulation-based design, uncertainty propagation, fragility analysis, etc., which require many numerical simulations including computational fluid dynamics (CFD) and fluid-structure interaction (FSI), are impractical because of their computational expense and long run times.

To mitigate the computational costs associated with CFD/ FSI models, machine learning (ML) and deep learning (DL) can be utilized to train models that represent the salient features of the numerical analysis with reduced runtimes. Once the model has been trained on the respective data, the ML models can run quickly, producing results in a shorter time than the numerical model. However, since the ML and DL algorithms are data-driven in nature, i.e., they may not enforce the governing equations of motion, their reliability and interpretability can be suspect in comparison to numerical simulations. The reliability and interpretability of the ML and DL algorithms make it difficult to trust their solutions for a given problem based on a purely data-driven approach. A number of studies have tried to address the interpretability of ML and DL algorithm (Gilpin et al. 2018) (Bibal and Frénay 2016) (Carvalho et al. 2019) (Moraffah et al. 2020). All concluded that ML and DL algorithms need to provide satisfactory explanations of their solution in order to be widely accepted.

This project addressed the interpretability and reliability of ML and DL algorithms by introducing physical equations into the learning algorithms. To understand ML and DL algorithms, this study produced results for a simple representation of a bridge using a single-degree-of-freedom (SDOF) oscillator subjected to static and dynamic analyses. Numerical results were calculated by using the finite element analysis framework Open System for Earthquake Engineering Simulation (OpenSees), which is often used for civil engineering applications. The numerical results from OpenSees were used to generate the training data. The residual, calculated by using OpenSees, was then incorporated into the loss function during training to aid learning of the desired equations of motion; the loss function is the main criterion from which the ML and DL algorithm learns and then evaluates the solution.

1.1. Objectives

This report presents a pilot study to use machine learning algorithms to learn static and dynamic structural response, which can be adapted for CFD/ FSI problems to better understand tsunami loading on bridges.

The objectives of this report included the following:

1. Review viable ML and DL algorithms and select potential algorithms for learning. There are many ML/DL algorithms available for different purposes. It is important to use the ML/ DL algorithm best suited to the problem for robust results. Many ML and DL algorithms have different characteristics; e.g., artificial neural networks (ANN) have been used for the design of steel structures (Adeli and Yeh 1989) and reliability analysis of steel frames (Papadrakakis et al. 1996), a convolutional neural network (CNN) can be used to identify damage by using images of structures (Cha et al. 2017) (Dung and Anh 2019), and a recurrent neural network (RNN) can be used for time series analysis of structures (Peng et al. 2021).
2. Develop a learning framework that allows viable ML/ DL algorithms selected from objective (1) to learn from the training data extracted from finite-element analyses while minimizing the residual to satisfy the governing equations. A proper framework is needed to include the physical equations into the ML/ DL algorithm to achieve improved reliability and interpretability of the ML/ DL results.
3. Test the ability of the ML/ DL algorithm to generalize to new conditions for the following metrics: accuracy (relative to the original finite element model), interpretability (maintaining the relevant physics), and dimensions (speed of computation). The data-driven ML/ DL algorithms tend to perform well only within the training data domain. This is because the ML/ DL algorithm conventionally learns the features of the data without attempting to satisfy the governing equations. A physics-informed ML/ DL algorithm should be able to generalize well to new conditions and extrapolate results beyond the training data, as it learns based on the residual as well as the data features.
4. Demonstrate the resulting ML/ DL architecture on static and dynamic SDOFs with and without physics.

1.2. Organization of Report

This report consists of seven chapters. Chapter 2 includes the literature review of relevant machine learning and deep learning algorithms. Chapter 3 presents the deep learning algorithm and its suitability for a static model with constant stiffness. Chapter 4 provides the deep learning algorithm implementation for a static model with varying stiffness. Chapter 5 introduces deep learning algorithms for dynamic problems. Chapter 6 describes a study that preliminarily estimates the effects of FSI on a flexible beam-column. Chapter 7 describes the summary and conclusions of the report.

2. MACHINE LEARNING/ DEEP LEARNING INTRODUCTION

2.1. Introduction

Machine learning (ML) is a branch of artificial intelligence (AI) that includes many algorithms to map inputs to outputs. When given enough data, ML identifies patterns and produces good approximations of the model representing that data. These approximations of the data may not be explainable/ interpretable, but the resulting ML program can still be used to identify patterns in the data (Goodfellow et al. 2016).

To conceptually understand ML, traditional analysis methods can be compared to ML techniques. Many analysis methods take inputs and then approximate outputs based on satisfying physical equations. For example, by Newton's second law, inertial forces arise when a body with mass is accelerated. In structural dynamics, the solution at the next time step can be analytically or numerically found based on adequately satisfying Newton's second law, given some inputs (e.g., properties of the body, like its mass, and an applied time-varying force $p(t)$) to estimate some outputs (e.g., the motion of the body in terms of displacement, velocity, and acceleration).

In contrast, ML takes inputs (e.g., properties of the body and $p(t)$) and outputs (e.g., the motion of the body) to approximate the program (e.g., finding a pattern representing Newton's second law); see figure 2.1. The ML algorithm extracts patterns from given inputs and outputs to produce the program rather than the output (Goodfellow et al. 2016). The resulting program from the ML algorithm then estimates new outputs from new inputs that were not in the original training data. This process of mapping given inputs to the outputs to learn the program is called learning or training. To check the performance of the ML program, the learned program is checked on an unseen dataset, and is known as testing.

2.2. Types of Machine Learning

Machine learning can be divided into two categories defined by the learning process: [i] supervised learning and [ii] unsupervised learning. These two types of machine learning differ from each other based on their application or task. A task is a process of how the algorithm estimates an output given some input from the program. Supervised learning is when the machine learning algorithm is given the inputs and outputs, and the class labels (or supervisor) are known. In contrast, unsupervised learning is when the machine learning algorithm knows only the inputs of the data and does not have any class labels.

2.2.1. *Supervised Learning*

In supervised learning, the model is assumed to be defined by parameters, θ:

$$y = g(x|\theta) \tag{2.1}$$

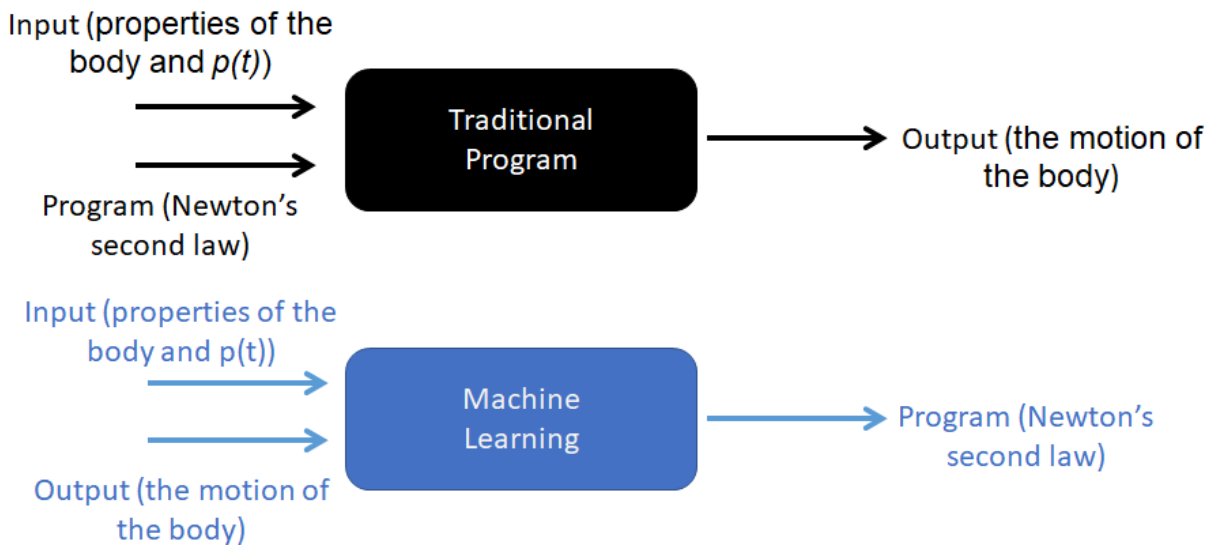

Figure 2.1: Comparison between traditional program and machine learning

where $g(.)$ is the model (algorithm/program) that the machine learning algorithm is attempting to learn; θ are the parameters that define the model (e.g., weights and biases explained in subsection 2.3.3), x is the inputs, and y is the outputs (or sometimes the class labels of the data).

Examples of supervised learning include regression problems. For example, in linear regression, Equation 2.1 becomes:

$$y = wx + w_0 \tag{2.2}$$

where w and w_0 are the weight and biases given to the inputs and are a variable in the learning process to get a better fit to the model.

Figure 2.2 shows an example of regression to predict the drift capacity of reinforced concrete walls. The inputs, x, include the reinforced concrete wall attributes such as axial load ratio, boundary longitudinal reinforcement ratio, web transverse reinforcement ratio, etc. and the outputs, y, include the drift capacity of the wall (Aladsani et al. 2022). Based on data pairs of (x, y), the supervised learning algorithm learns y as a function of x that follows the form of Equation 2.2 (Alpaydin 2020).

2.2.2. *Unsupervised Learning*

The main aim of unsupervised learning is to find patterns in the input data (Alpaydin 2020). Finding clusters or groups in the data is an example of unsupervised learning. For example, unsupervised learning can be used to find groups or patterns within the data for damage detection in structures (Daneshvar and Sarmadi 2022).

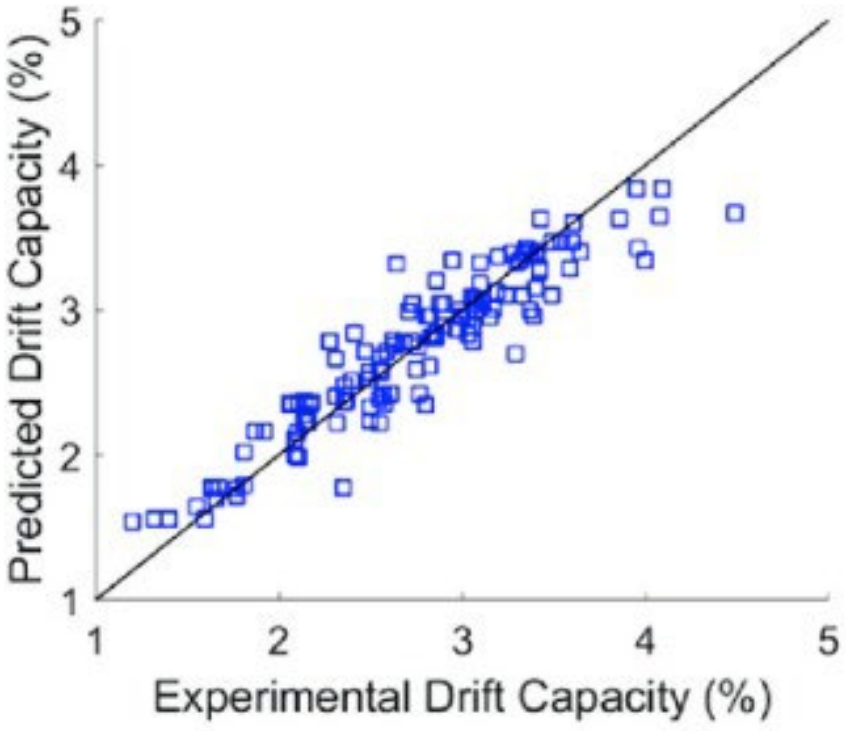

Figure 2.2: Supervised learning regression example: drift capacity for reinforced concrete walls predicted vs experimental (Aladsani et al. 2022)

2.3. Deep Learning

Deep learning is a subcategory of machine learning inspired by the structure and function of the brain's neural network. Originally, deep learning was known as the multi-layer perceptron (MLP) (Rosenblatt 1958). Over the years, deep learning has been defined by many different names, including deep learning models, nets, neural nets, or neural networks. Research interest in deep learning grew in the 1980s (Parker 1985) (LeCun et al. 1988) and then peaked in the 2000s due to the availability of massive amounts of data (Nichols et al. 2019).

2.3.1. Artificial Neural Network

An artificial neural network (ANN) is composed of a collection of connections known as neurons. Initially, neurons were modeled in 1943 (McCulloch and Pitts 1943) as a switch that receives information from other neurons. Depending on the relevance of the information received, the neurons remain active or inactive. In modern ANNs, these neurons are organized primarily into three types of layers, namely the 1) input layer, 2) hidden layer(s), and 3) output layer; see figure 2.3. The properties of the ANN and its function depend on the properties of the hidden layers. For example, if the hidden layers are convolution

layers, the network becomes a convolution neural network (CNN). If the hidden layers are long short-term memory (LSTM) layers, the network becomes a recurrent neural network (RNN). The hidden layers can also be dense layers, meaning all the neurons of the network are connected.

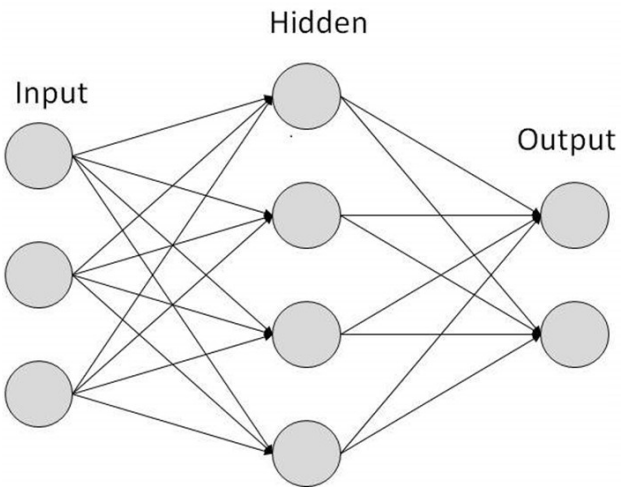

Figure 2.3: Architecture of an ANN

Artificial Neuron

Neurons are the fundamental units of the brain. The brain consists of 100 billion biological neurons that perform a simple operation by receiving electrical pulses from other neurons. They receive information (inputs) from the external world, send commands (outputs) to other parts of the body, e.g., muscles, and relay or transform electrical signals. A brain neuron consists of dendrites, soma (cell body), and axons (figure 2.4).

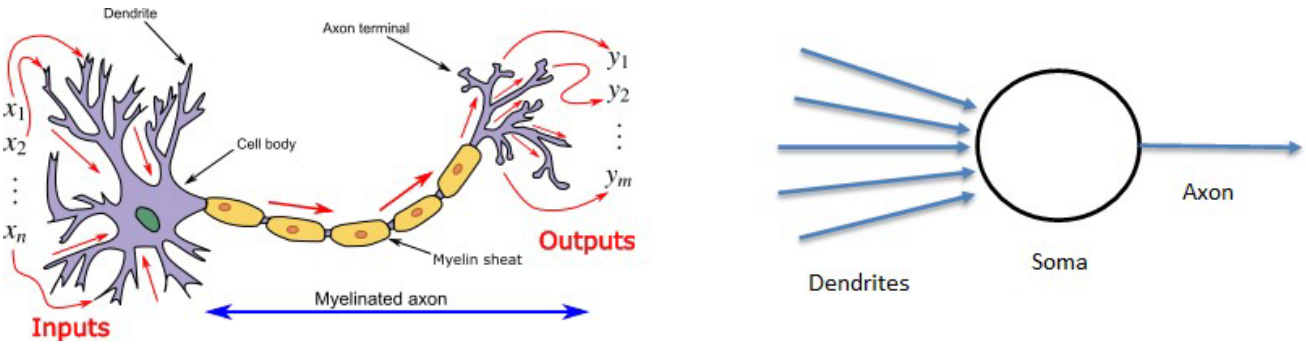

Figure 2.4: Brain neuron

An artificial neuron is roughly based on the function of brain neurons. In ANNs, the functionality of the neuron is based on the activation function.

2.3.2. *Activation Function*

The activation function introduces nonlinearities into Equation 2.2, making the network capable of learning complex problems. There are many activation functions available today, and each activation function has advantages and disadvantages. The selection of the activation function depends on nonlinearity (e.g., nonlinear neuron captures parabolic trends), range (e.g., <0,1>,<-1,1>, or <0,∞>), derivative (useful for back propagation, explained in subsection 2.5.2), and value near the origin (Gulikers 2018). Some of the most used activation functions and their advantages and disadvantages are summarized below.

Sigmoid

The sigmoid function is the most widely used activation function. It is defined as:

$$Sigmoid(x) = \frac{e^x}{1 + e^x} \tag{2.3}$$

where x is the input to the function.

The advantage of the sigmoid function is that it has a smooth gradient and maps to values between 0 and 1, resulting in clear predictions. The sigmoid function predictive qualities are illustrated in figure 2.5,where the values of x above 2 and below -2 are mapped to values of 1 and 0 respectively.

A disadvantage of the sigmoid function is its vanishing gradient, i.e., very large and small values of x result in a negligible change in $f(x)$. In this case, the NN learns very slowly or does not learn at all. Other disadvantages of the sigmoid function include outputs that are not centered around zero and the computational expense.

Hyperbolic Tangent

The hyperbolic tangent (tanh) function is a scaled version of the sigmoid function. It is defined as:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.4}$$

Note, $f(x)$ is now zero-centered. However, the disadvantages of having a vanishing gradient and being computationally expensive, similar to the sigmoid function, still remain. Figure 2.6 shows the hyperbolic tangent function. Figure 2.7 compares the performance of tanh and sigmoid functions, showing that tanh performs better than sigmoid (Glorot and Bengio 2010).

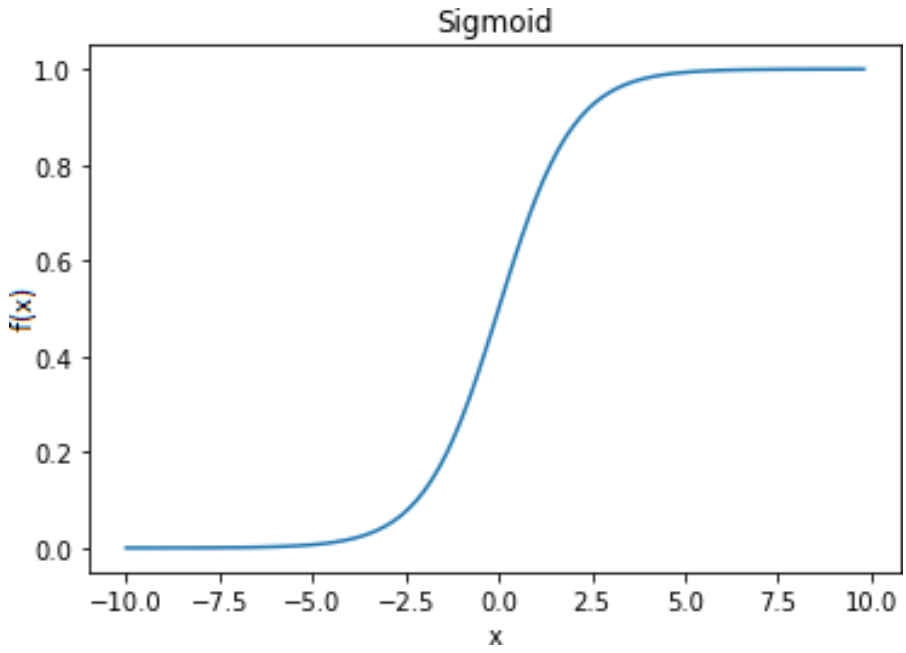

Figure 2.5: Sigmoid function

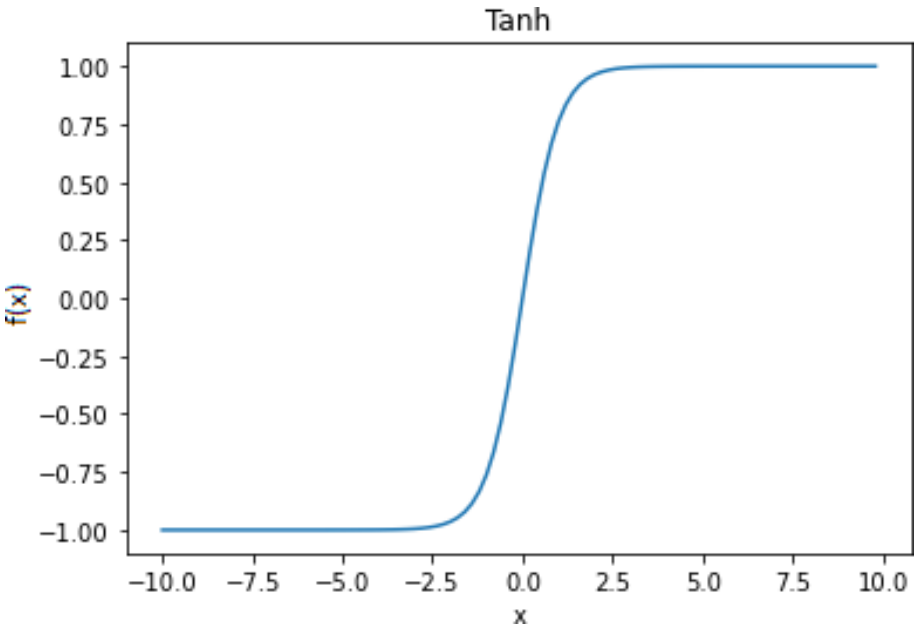

Figure 2.6: Hyperbolic tangent function

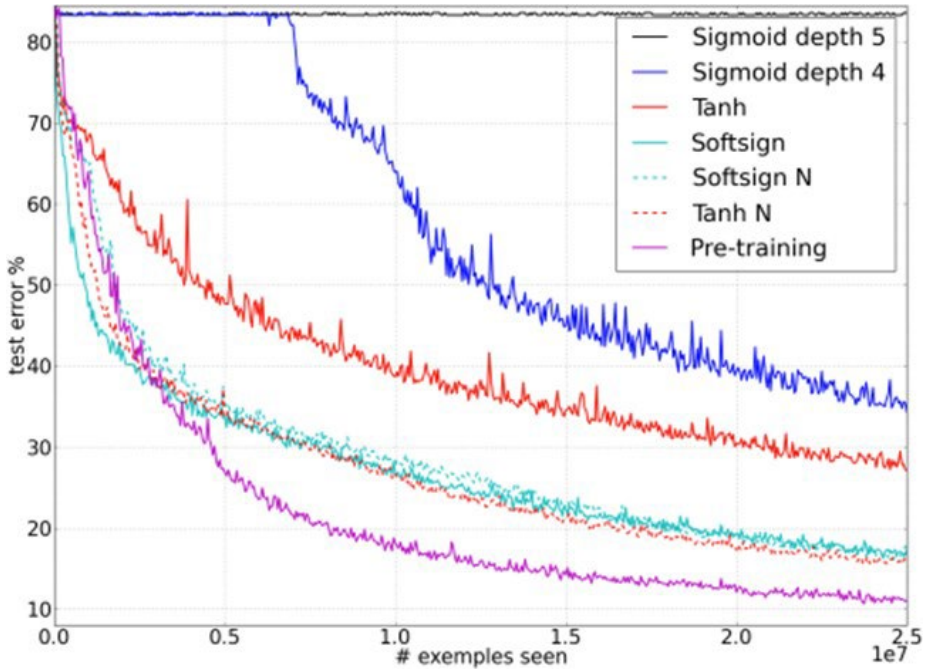

Figure 2.7: Hyperbolic tangent (red) has a lower test error than sigmoid (blue) (Glorot and Bengio 2010)

Rectified Linear Unit

The Rectified Linear Unit (ReLU) is a popular activation function. The equation for ReLU is

$$ReLU(x) = max(0, x) \tag{2.5}$$

The ReLU function always maps to zero for all negative inputs. When the input is greater than zero, the output is the identity function. The derivative for positive input values of x is always equal to 1 (figure 2.8), making the network computationally efficient and faster at learning. The convergence of ReLU activation functions can be seven times faster than the sigmoid function (Krizhevsky et al. 2012) (figure 2.9).

Disadvantages of the ReLU function include the dying ReLU phenomenon, which is when inputs are nearly zero or negative. In this case, the gradient becomes zero and the network cannot learn anymore.

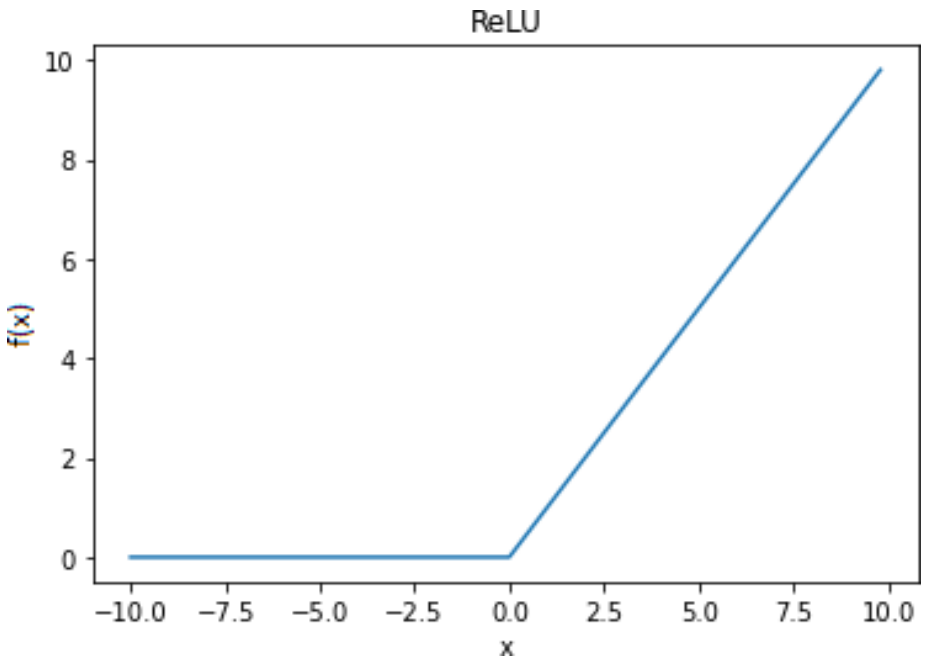

Figure 2.8: ReLU function

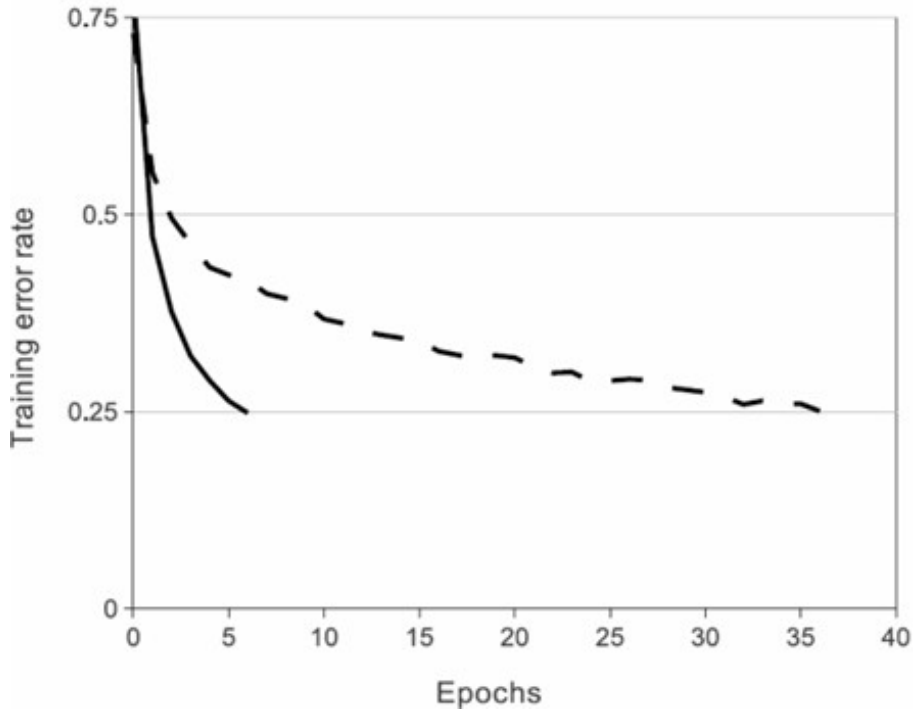

Figure 2.9: ReLU vs sigmoid function

Leaky ReLU

To avoid the dying ReLU phenomenon, the leaky ReLU was introduced as an activation function. It is defined as:

$$LeakyReLU(x) = max(ax, x) \tag{2.6}$$

where a is a small constant. Figure 2.10 plots the leaky ReLU for an a value of 0.05.

Although the leaky ReLU solves the dying ReLU problem, the predictions of leaky ReLU may not be consistent for negative input values of x.

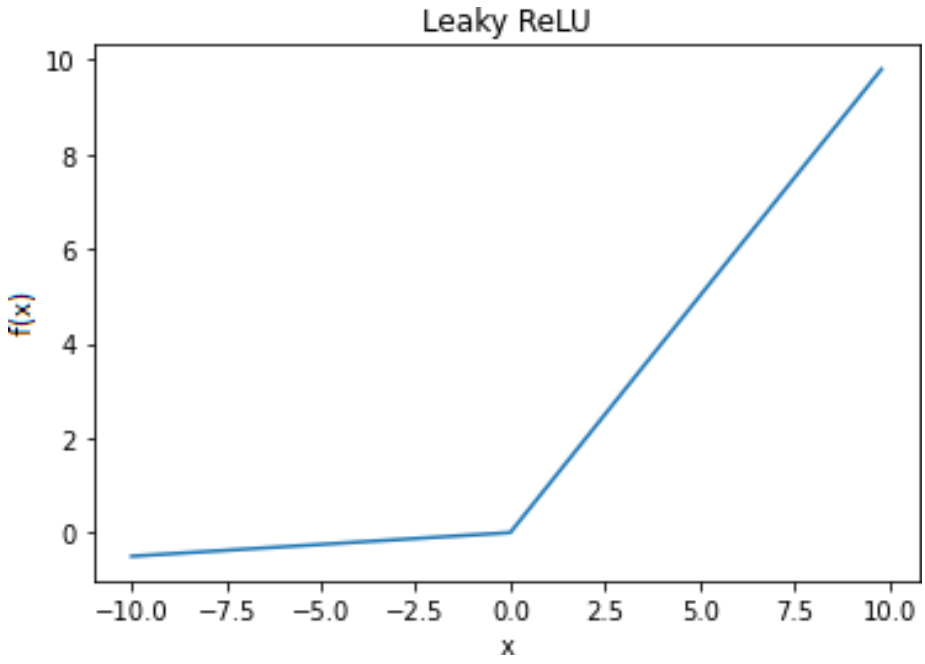

Figure 2.10: Leaky ReLU function

2.3.3. Weights and Biases

A neural network is the combination of connected layers of neurons. To ”turn on” each neuron, multiple weights, w, are assigned to the inputs. The dot product of the weights w and inputs x are calculated, and an activation function is applied to generate the output. A bias, b, provides flexibility (shifting the results towards positive and negative) in the neuron. Using these terms, Equation 2.1 can be re-written as

$$h(x) = g(x_1 w_1 + x_2 w_2 + x_3 w_3 + b)$$
$$h(x) = g(\boldsymbol{w}^T x + B) \tag{2.7}$$

where $g(.)$ is the activation function, x are inputs, w are weights assigned to the inputs, b is the bias, and $h(x)$ is the output.

Increasing the weights of the input activates the function more to reflect the importance of the input. Figure 2.11 (a) shows that an increase in weights to the sigmoid function makes the curve more linear and has a higher activation,whereas figure 2.11 (b) shows that an increase in bias changes the node activation from -4 to 0 to 4 to add more flexibility to the function.

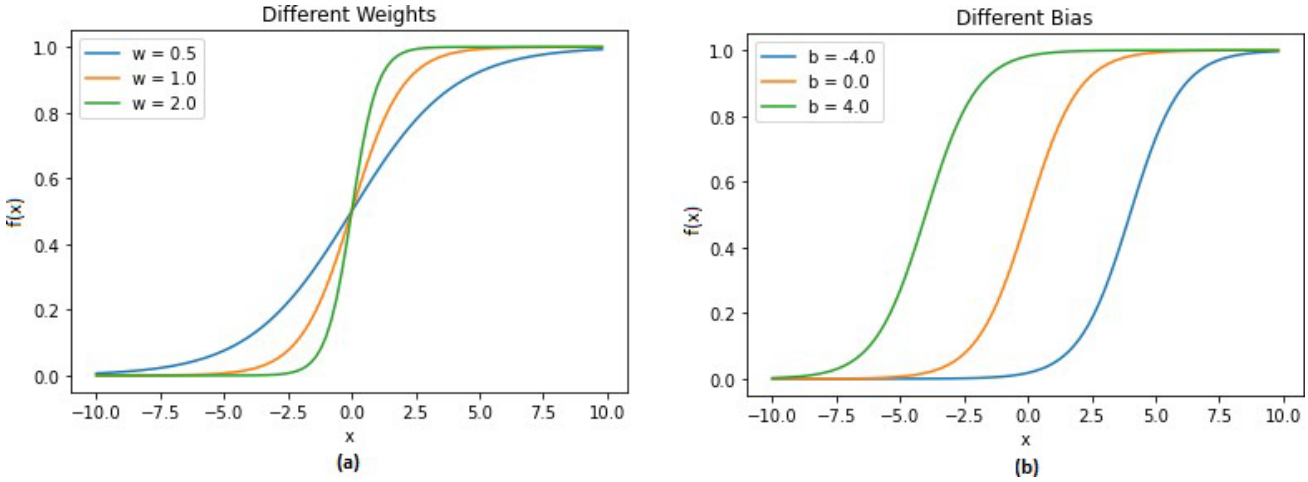

Figure 2.11: (a) Different weights, **(b)** Different bias

2.3.4. *A Simple Neural Network*

A neural network becomes more flexible as more hidden layers are added to the network. With no hidden layers in the neural network, the program collapses into linear regression in Equation 2.2. Adding more hidden layers to the NN increases the nonlinearity of the network. A single hidden layer network acts as the boundary of a convex region, as shown in figure 2.12. It can estimate the XOR problem, i.e., if the two inputs of the problem are different, then the output is true or else the output is false. For example, if the inputs are 1 and 0, then it is classified as true, but if the inputs are 0 and 0 or 1 and 1, then the output is false.

When the NN has two hidden layers, the flexibility of the network increases, and the NN is then the combination of convex regions, as shown in figure 2.13.

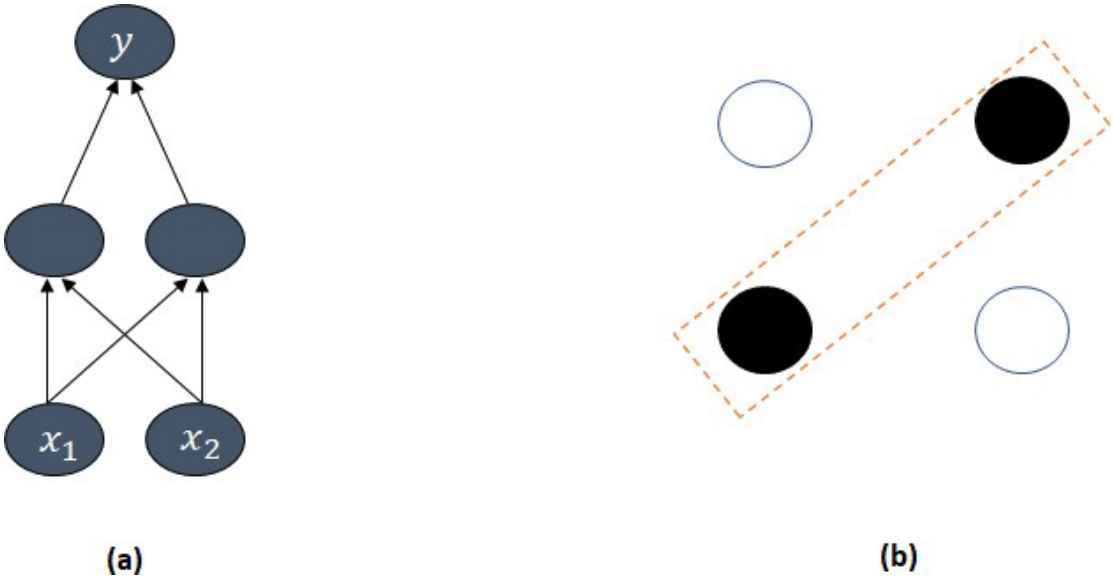

Figure 2.12: (a) A single hidden layer NN, **(b)** XOR problem

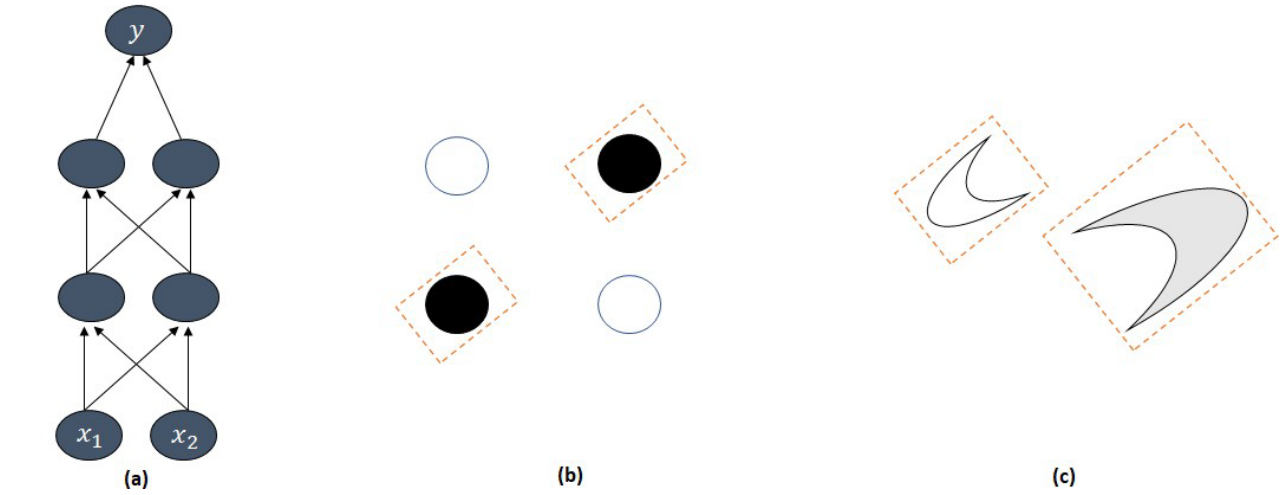

Figure 2.13: (a) Two hidden layer NN, **(b)** XOR problem, **(c)** Complex problem

2.4. Feedforward Pass

The information in an NN flows from the inputs, x, to the outputs by computing the function defined in Equation 2.1. This flow of information is known as the feedforward pass (Goodfellow et al. 2016). The purpose of a feedforward pass is to find the model that can be used to estimate the outputs from the inputs. This model is represented by weights, w, assigned to each node of the NN and the bias,

b.

The feedforward pass can be explained with the help of a three-layer network, which includes an input, one hidden, and an output layer; see figure 2.14. The outputs of each layer, h, are described mathematically as:

$$
\begin{aligned}
h_1^{(2)} &= g(x_1 w_{11}^{(1)} + x_2 w_{12}^{(1)} + x_3 w_{13}^{(1)} + b_1^{(1)}) \\
h_2^{(2)} &= g(x_1 w_{21}^{(1)} + x_2 w_{22}^{(1)} + x_3 w_{33}^{(1)} + b_2^{(1)}) \\
h_3^{(2)} &= g(x_1 w_{31}^{(1)} + x_2 w_{32}^{(1)} + x_3 w_{33}^{(1)} + b_3^{(1)}) \\
h_{W,b}(x) &= h_1^{(3)} = g(h_1^{(2)} w_{11}^{(2)} + h_2^{(2)} w_{12}^{(2)} + h_3^{(2)} w_{13}^{(2)} + b_1^{(2)})
\end{aligned}
$$

where $w_i^{(k)}$ refers to i node in the connection layer, j refers to the originating layer, and k refers to the number of layers.

The output of each layer is the summation of the bias, b, and inputs, x, of the previous layer multiplied by a weight. This summation is then passed through an activation function, g, to introduce nonlinearities.

To initiate training, the weights and the bias at the start of the training of the NN are initially set to random values. These values are then updated by backpropagation (subsection 2.5.2), with each feed-forward pass, updating the "model" represented by the NN.

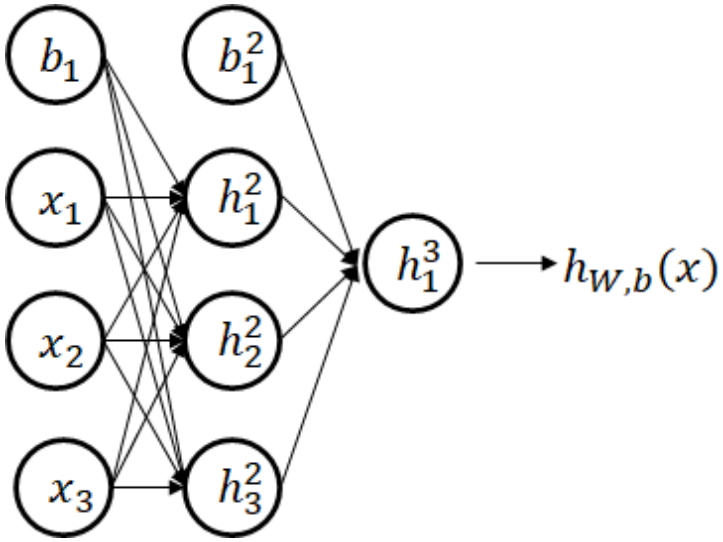

Figure 2.14: Neural network with input, hidden and output layer

2.5. Training a Neural Network

To train a NN for an estimate of the output, the NN should identify the best weights and bias to represent the problem. These weights and biases are optimized by the loss function (error) at the end of a feedforward pass.

2.5.1. Cost Functions

In supervised learning, the loss function is minimized to find optimal weights and bias. The loss function returns the value of the error to the training data after each feedforward pass. The cost function is then the average of the loss function over n training samples. The cost function is reduced by checking the input and output pairs of known data and varying the weights accordingly to minimize the cost function. Thus, the goal of the cost function is to provide a metric to measure increases or decreases in the performance of the network. When the cost function reduces to a tolerance defined by the user, the algorithm is said to be converged. The basic form of the cost function is

$$C(\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y}) \tag{2.8}$$

where $\boldsymbol{w}$ are the weights of the NN, $\boldsymbol{b}$ is the bias, $\boldsymbol{x}$ is the input to the network and $\boldsymbol{y}$ are the expected outputs in vector forms.

There have been many proposed loss functions with different properties. All of these have the following two important attributes (Nielsen 2015) (Papalambros and Wilde 2000):

- The cost function should be a scalar value representing the error of n training samples with respect to the data. This is important to compute the derivative of the cost function, which is needed to minimize the cost function.
- The input-output layers should remain unchanged throughout the optimization process, and the cost function should only be calculated from the outputs of the network.

Quadratic Cost Function

The quadratic cost function, more commonly known as mean-squared error (MSE), is often used in statistical problems. MSE is the minimum average squared distance value between the estimated values and the original values. The NN tries to minimize the cost function to achieve better performance based on predicted values of output and the original outputs from the training data. However, MSE does not perform well in combination with some activation functions, such as the sigmoid function (Gulikers

2018). Thus, MSE is used for regression problems. MSE is written as:

$$C_{MSE}(\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{n} \sum_{i=1} (\boldsymbol{y}^* - \boldsymbol{y})^2 \tag{2.9}$$

where n is the number of training points and $\boldsymbol{y}^*$ is the estimated values from the model in vector form.

Cross Entropy Cost Function

The cross-entropy cost function enhances the performance of the NN. Cross-entropy reduces the effect of slow learning, which can occur due to saturation using the MSE; i.e., slight differences between the expected and original values result in large MSE values. Cross-entropy is defined as:

$$C_{CE}(\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y}) = -\frac{1}{n} \sum [\boldsymbol{y} \ln \boldsymbol{y}^* + (1 - \boldsymbol{y}) \ln(1 - \boldsymbol{y}^*)] \tag{2.10}$$

L1 Cost Function

The L1 cost function or mean absolute error (MAE) is the absolute difference between the estimated value and the original value. It has the same properties as MSE in terms of saturation causing slow learning, but it is more reliable as the errors are not as large as in MSE. The L1 cost function is written as:

$$C_{MAE}(\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{n} \sum_{i=1} (\boldsymbol{y}^* - \boldsymbol{y}) \tag{2.11}$$

2.5.2. Back Propagation

Outputs of the NN are predicted from the inputs during the feed-forward pass. These predicted outputs and the original training outputs are used to calculate the cost function. Back propagation is used to adjust the weights of the layers to minimize the loss function.

Back propagation uses the chain-rule to estimate this change in the cost function due to the estimates for the weights. For example, referring to figure 2.14, back propagation calculates the change in the cost function due to the weight assigned to the second neuron in the second layer ($w_{12}^{(2)}$); i.e. to estimate the change of the cost function C with respect to $w_{12}^{(2)}$, back propagation calculates the change of C with respect to the output $h_1^{(3)}$, the change of $h_1^{(3)}$ with respect to activation function in the second layer second neuron $z_1^{(2)}$, and the change of $z_1^{(2)}$ to $w_{12}^{(2)}$. To calculate the change in the cost function with

respect to the weight of one neuron, $w_{12}^{(2)}$, these results are multiplied based on the chain rule:

$$\frac{\partial C}{\partial w_{12}^{(2)}} = \frac{\partial C}{\partial h_1^{(3)}} \frac{\partial h_1^{(3)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}} \tag{2.12}$$

The last term of Equation 2.12 is:

$$h_{w,b}(x) = h_1^{(3)} = g(h_1^{(2)} w_{11}^{(2)} + h_2^{(2)} w_{12}^{(2)} + h_3^{(2)} w_{13}^{(2)} + b_1^{(2)})$$

$$h_{w,b}(x) = h_1^{(3)} = g(z_1^{(2)})$$

$$z_1^{(2)} = h_1 w_{11}^{(2)} + h_2 w_{12}^{(2)} + h_3 w_{13}^{(2)} + b_1^{(2)}$$

$$\frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}} =$$

The second to last term of Equation 2.12 depends on the activation function used in the layer of the NN. Assuming that the sigmoid activation function is used:

$$\frac{\partial h_1^{(3)}}{\partial z_1^{(2)}} = g'(z) = g(z)(1 - g(z))$$

The first term of Equation 2.12 is the derivative of the cost function with respect to the output. Assuming we use the quadratic function or MSE as the cost function (Equation 2.9) and, for simplicity, using n = 2:

$$C(\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2} \| y_1 - h_1^{(3)}(z_1^{(2)}) \|^2$$

$$Let\ u = \| y_1 - h_1^{(3)}(z_1^{(2)}) \| \ so\ C = \frac{1}{2} u^2$$

$$\frac{\partial C}{\partial h_1^{(3)}} = -(y_1 - h_1^{(3)})$$

These three terms are multiplied to give the change of cost function with respect to the weight of the second neuron in the second hidden layer. The above calculations can be simplified by defining δ as a new term:

$$\delta_i^{(n^l)} = -(y_i - h_i^{(n_l)}) g'(z_i^{(n_l)})$$

where i is the node number of the output layer and for the example case of figure 2.14, is always 1. $h_i^{(n^l)}$ is the output from the final layer.

The cost function in terms of δ and the output of its layer is:

$$\frac{\partial C}{\partial w_{ij}^{(l)}} = h_j^{(l)} \delta_i^{(n_l)} \tag{2.13}$$

The δ term needs to be back propagated to ultimately connect the weight of each neuron in a layer of the NN to the cost function. A similar process is used to back propagate the bias. Equation 2.13 is then used in an optimization procedure, typically gradient descent, to update new weights and biases for the NN based on the minimization of the cost function. The new weights and biases for the NN are:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \frac{\partial C(\boldsymbol{w}, \boldsymbol{b})}{\partial w_{ij}^{(l)}} \tag{2.14}$$

$$b_i^{(l)} = b_i^{(l)} - \frac{\partial C(\boldsymbol{w}, \boldsymbol{b})}{\partial b_i^{(l)}} \tag{2.15}$$

2.5.3. *Optimization of Neural Network*

The weight and biases are optimized based on the loss function so that the network learns the patterns from the input and output pairs. The main function of optimization is to minimize the loss function for its weights and biases. For example, Equation 2.16 is based on optimization of the weights using gradient descent.

$$\begin{aligned} &\min_{w} C(\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y}) \\ &while \, \|\nabla w\| > tolerance \\ &\mathbf{w} = \mathbf{w} - a\nabla w \end{aligned} \tag{2.16}$$

where a is the learning rate, i.e., the step size taken towards convergence.

The optimization problem tries to find the weights and biases so that the cost function of the network is at a local minimum. Note, the local minimum is sometimes the near-best solution, as finding

the global minimum can be hard and at times unnecessary (Goodfellow et al. 2016). This is done by moving in the opposite direction of the slope from the current point by α (figure 2.15).

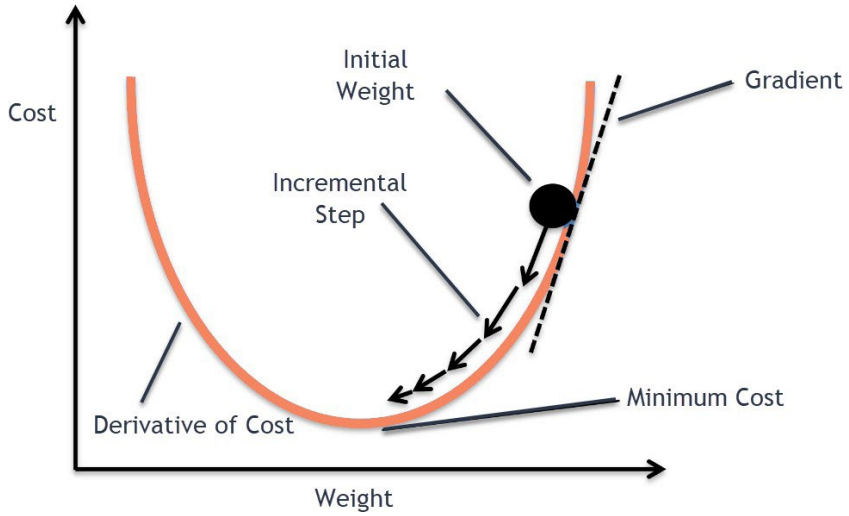

Figure 2.15: Gradient descent algorithm (M 2020)

The optimization problem is convex for a single node NN. This means that all the local minima are also the global minima, and there is only one solution. But as more neurons are added to the NN, the optimization problem becomes non-convex, and there can be multiple minima, also known as saddle points. As the gradient tends towards zero near these saddle points, the gradient descent algorithm can get stuck in one of these local minimas and result in sub-optimal weights and biases (figure 2.16).

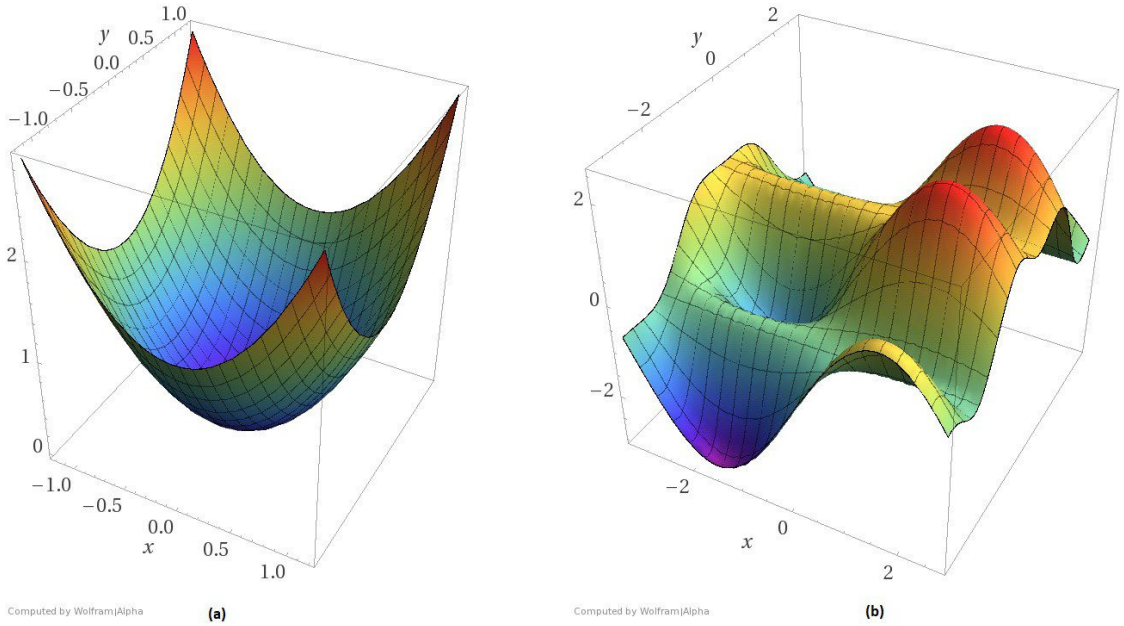

Figure 2.16: (a) Convex, **(b)** Non-convex (Zadeh 2016)

There is no straightforward solution to selecting a local minimum based on the non-convex nature of NNs. To enhance training, the following strategies have been proposed: [i] stochastic gradient descent optimization, [ii] gradient descent optimization with momentum, [iii] Adam optimization, [iv] dropout, [v] data and batch normalization, and [vi] regularization. Some of these techniques are related to the optimization of the gradient descent algorithm,whereas others are related to the regularization of data and the network.

Stochastic Gradient Descent

In gradient descent, when the NN has many nodes in the hidden layers, the gradient ∇W in Equation 2.16 can be computationally expensive and take a significant amount of time to compute. The gradient computation makes the NN slow to optimize the weights and bias for a batch, which is the entire training set. Also, the network can get stuck near a saddle point or at a non-optimum local minimum.

Additionally, computing the gradient over a batch can lead to memory issues, as the gradients taken for a large network can be very large.

These problems can be mitigated by using mini-batches,where gradient descent is calculated over smaller portions of the data in an optimization algorithm known as a stochastic mini-batch approximation or stochastic gradient descent (Equation 2.17). The NN learns by re-arranging the inputs randomly to make mini-batches. An iteration is defined as the computation of the gradient over a mini-batch. The process of randomly selecting different mini-batches from the training data is then repeated several times. Each time an input example is looped through the NN, one cycle through the entire training dataset, a number of iteration (epoch) is recorded (figure 2.17). Many epochs may be required to optimize the NN.

$$\min_{w} \sum_{i=1}^{n} C(f(x_i, \mathbf{w}), y_i)$$
$$\nabla_w \sum_{i=1}^{n} \frac{\partial C(f(x_i, \mathbf{w}), y_i)}{\partial w} \tag{2.17}$$

Momentum

The weights and bias of the network are updated by minimizing the loss; e.g., using the gradient descent algorithm. For the gradient descent algorithm to converge, the learning rate α in Equation 2.16 needs to be small enough. However, learning rates too small can result in slow convergence. Generally, a large learning rate is used initially. If the algorithm fails, the learning rate is decreased by a factor of 10, and so on.

Momentum can be used to enhance the speed of learning (Moreira and Fiesler 1995). Momentum

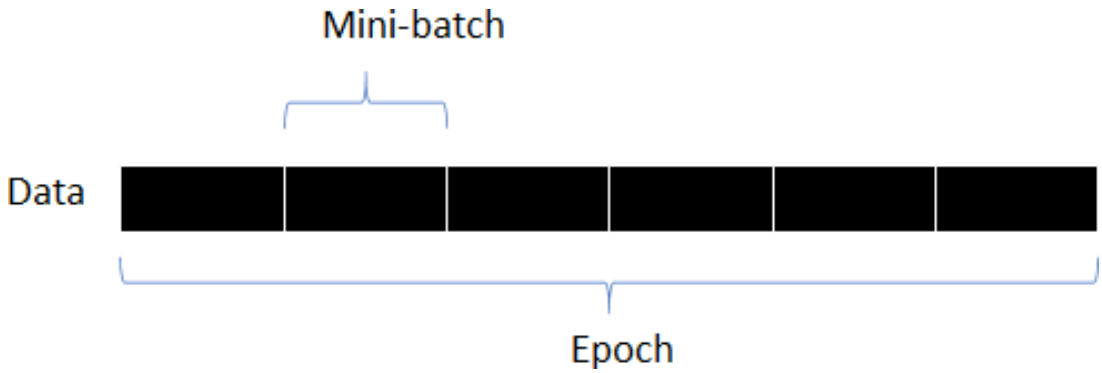

Figure 2.17: Epoch vs mini-batch

includes the gradient of the function in the previous step as a part of the update to the next step.

Algorithm 1: Gradient Descent with Momentum

INITIALIZATION: $D_0 = 0$
while $\|\nabla f(\mathbf{w})\| >$ tolerance **do:**
Calculate $D_{t+1} = \mu D_t - \alpha G_t$
$W_{t+1} = W_t + D_{t+1}$
end while.
where: α is the learning rate, μ is momentum, W are the weights

Momentum decreases the time required to reach convergence, by giving better results as it uses previous information to find local minima; see figure 2.18. Values representing the contribution of momentum (μ) ranges from 0 to 1.0 and are typically between 0.6 to 0.8. If the momentum is equal to zero, then the algorithm collapses to gradient descent without momentum.

Adam

Adam (adaptive moment estimation) is another commonly used optimization technique. Adam combines the advantages of two other optimization techniques in gradient descent, namely AdaGrad and RMSprop (Kingma and Ba 2014) (describing these is outside the scope of this report). Adam stores the exponentially decaying average of the past squared gradients and past gradients. This allows Adam to behave like a heavy ball with friction (Heusel et al. 2017), allowing the optimization algorithm to shoot over non-optimal local minima to find more optimal minima. For example, figure 2.19 shows the heavy ball with the friction concept of Adam,where it shoots over θ^+ and settles at minimum θ^*.

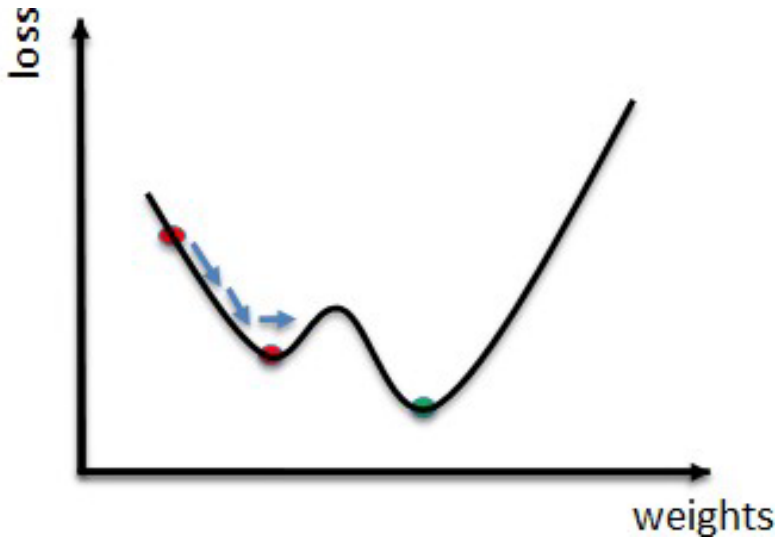

Figure 2.18: Gradient descent with momentum. Shifting to a better minimum

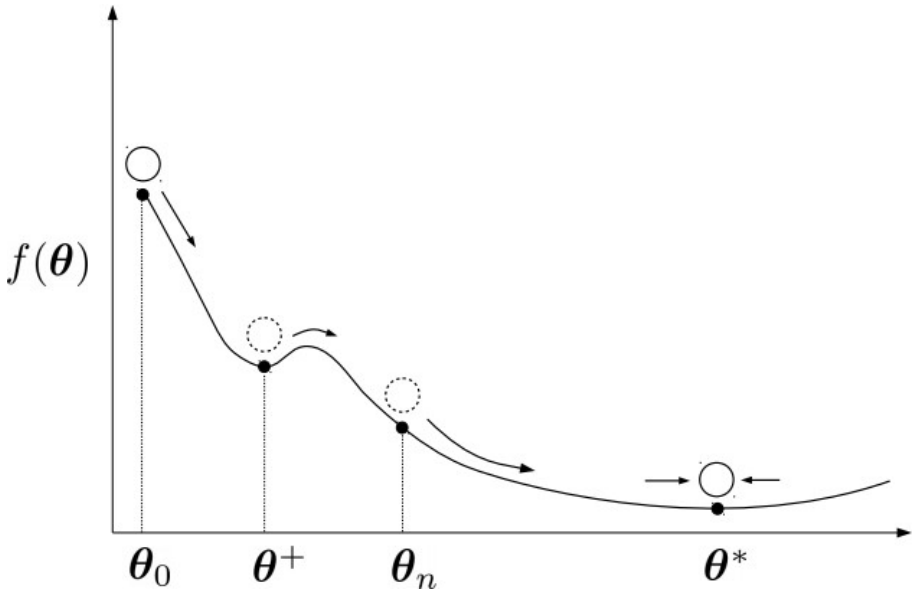

Figure 2.19: Adam: heavy ball with friction

Dropout

A deep network can sometimes cause problems as it can overfit the data. Overfitting is when the NN learns the trends in the training set but, when given new data points, is unable to generalize those trends to new data. In contrast, underfitting occurs when the network is unable to learn the general trends in the training data set.

Overcoming underfitting and overfitting requires a trade-off between bias and variance. A bias is a difference between the average prediction of the model and the correct value. A high bias indicates little attention to data (high error). Variance is the variability and spread in the model. A high variance means the generalization of the model. Figure 2.20 shows the underfitting and overfitting zones for high bias and high variance and how they affect the training and test error of the NN.

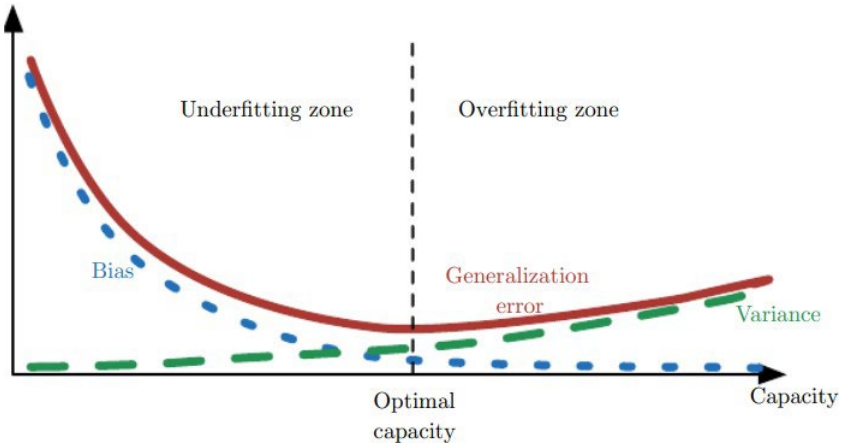

Figure 2.20: Underfitting vs overfitting (Goodfellow et al. 2016)

Dropout can alleviate the overfitting problem by ignoring neurons in the NN if they become co-dependent during the training phase. The NN then becomes smaller, and more appropriate weights can be assigned to the remaining neurons, while neurons that might contain outliers can be neglected (figure 2.21).

Data and Batch Normalization

Training of the NN depends on data structuring. If the data are unscaled, one value of the inputs can dominate the other values, and the NN will assign more weight to this input value. The gradient would “explode” in the gradient descent algorithm, as one weight would be significantly larger than the other weights, causing performance issues in the NN. Therefore, normalizing the data so that all points are of a similar range/scale can ensure that all inputs are treated similarly. Normalization also reduces the training phase of the NN.

Sometimes, even after normalizing the data, one of the weights of the NN can still become large. In very deep NNs, which have several layers and functions, assigning or updating weights to each layer

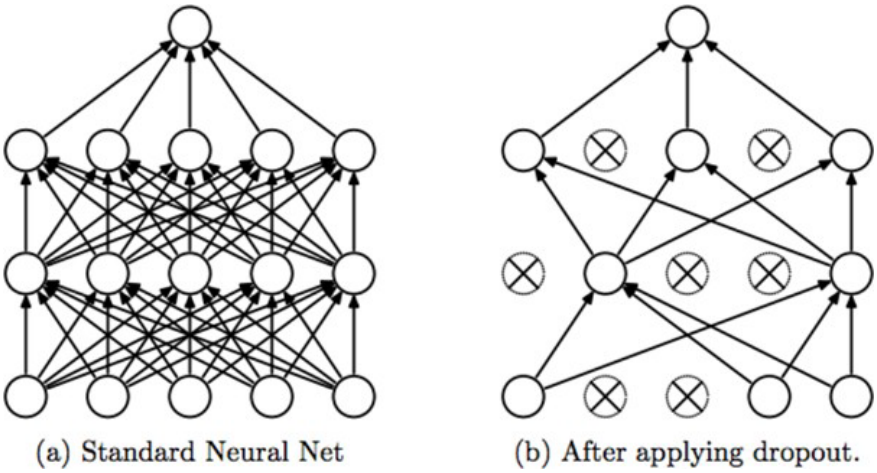

Figure 2.21: Dropout to prevent overfitting (Srivastava et al. 2014)

is done simultaneously. The updates assume that the functions remain constant, but in the NN these functions are changed simultaneously, which can cause issues (Goodfellow et al. 2016). Batch normalization can mitigate these issues.

Batch normalization re-parameterizes the NN. The output from the activation function is normalized. Batch normalization is done by multiplying the output by an arbitrary parameter g and adding arbitrary parameter b. This procedure sets a new mean and standard deviation for the data and is optimized during the training phase.

Regularization

Another way to address overfitting is to introduce a regularizer to the cost function. This regularizer term penalizes the loss function to aid in generalization. An example of a regularizer in the case of weight decay which is added to Equation 2.9 is:

$$C_{MSE}(\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{n} \sum_{i=1}^{\mathbf{I};} (\boldsymbol{y}^* - \boldsymbol{y})^2 + \lambda \boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{w}) \tag{2.18}$$

The additional regularizer term at the end of Equation 2.18 is then minimized, as well as the MSE cost function, by the NN during learning. This approach of adding a regularizer to the cost function is known as regularization (Goodfellow et al. 2016).

2.5.4. *Training NN Using Gradient Descent*

The following algorithm summarizes the steps of training a NN by using the gradient descent algorithm.

Algorithm 2: Training NN Using Gradient Descent

INITIALIZATION: Randomize values of weights for each layer in the NN
while iterations < iteration limit **do:**
Set $\Delta \boldsymbol{w}$ and $\Delta \boldsymbol{b}$ to random values
For samples 1 to m:
a. Perform a feed forward pass through all the layers. Store the activation function outputs $h^{(l)}$
b. Calculate the $\delta_j^{(nl)}$ value for the output layer
c. Use back propagation to calculate the $\delta^{(l)}$ values for layers 2 to $(n-1)$
d. Update the ΔW and Δb for each layer
Perform a gradient descent step using

$$w^{(l)} = w^{(l)} - a\left[\frac{1}{m}\Delta w\right]$$
$$b^{(l)} = b^{(l)} - a\left[\frac{1}{m}\Delta b\right]$$

end while.
where: a is the learning rate, m are training samples.

---

3. STATIC MODEL WITH CONSTANT STIFFNESS

3.1. Introduction

Engineering applications for ML include solving partial differential equations (Sirignano and Spiliopoulos 2018), fluid dynamic problems (Raissi, Yazdani, et al. 2018) (Sanchez-Gonzalez et al. 2020) (White et al. 2019b), emulating physical systems (Beucler et al. 2019), and structural health monitoring (Flah et al. 2021), among others. However, ML algorithms do not use scientific theory but instead are data-driven and find a program based on the pattern of mapped inputs to the outputs. In contrast, traditional programs or physics-informed models in numerical methods often are based on scientific theory (figure 3.1). To enhance the reliability and robustness of the ML results, physical equations can be introduced into the ML algorithm, referred to as physics-informed neural networks (PINN) (Raissi, Perdikaris, et al. 2019), physics guided neural networks (PGNN) (Karpatne et al. 2017) (Figure 3.1), or scientific computational with artificial neural networks (SciANN) (Haghighat and Juanes 2020).

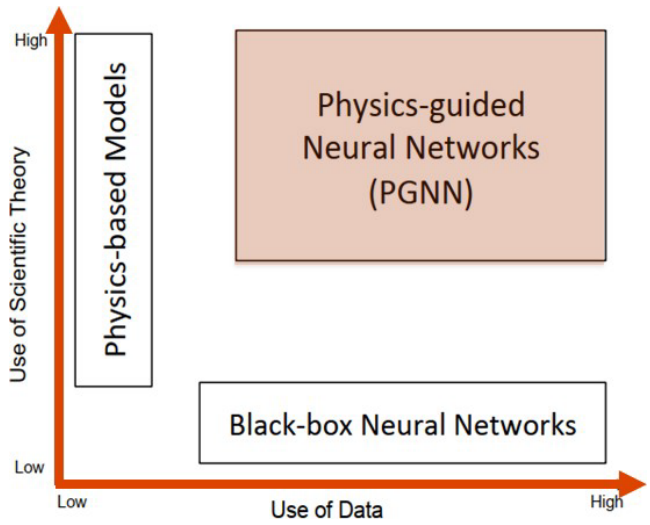

Figure 3.1: Physics-informed neural networks

The literature showed that there are many ways of incorporating physics into the NN. The most common method is to introduce the residual into the loss function of the NN algorithm. The addition of the residual into the loss function helps the learned model to be consistent with the physical equations of the problem. NN loss-based physics models can also better extrapolate and generalize outside the original data domain.

The applications of NN loss-based physics-informed models include solving partial differential equations (PDEs) (Y. Zhu et al. 2019) (Geneva and Zabaras 2020), discovering governing equations (Loiseau

and Brunton 2018) (Doan et al. 2019), inverse modelling (Raissi, Yazdani, et al. 2018) (Kahana et al. 2020), parameterization (Linfeng Zhang et al. 2018) (Beucler et al. 2019), down-scaling (Esmaeilzadeh et al. 2020) (Bode et al. 2021), uncertainty quantification (Y. Yang and Perdikaris 2018) (Y. Yang and Perdikaris 2019) (Y. Zhu et al. 2019) (Geneva and Zabaras 2020) (Karumuri et al. 2020) (L. Yang et al. 2019), and generative models (J.-L. Wu et al. 2020) (Shah et al. 2019).

Another type of NN physics-informed method is referred to as hybrid physics NN models. Hybrid physics NN models combine the physics-informed model (such as a finite element model) with an NN model. Hybrid physics NN models are often used for residual modelling (Forssell and Lindskog 1997) (Thompson and Kramer 1994) (San and Maulik 2018) (Kani and Elsheikh 2017), in which the output of the physic-based model is used as input to an NN model (Karpatne et al. 2017), replacing part of physic-based model with the NN (Parish and Duraisamy 2016) (Liang Zhang et al. 2019); e.g., combining results from the physics models and NN model (Chen et al. 2018) (Paolucci et al. 2018), as well as using the NN to refine the inversion models obtained from physics models in inverse modelling (Bubba et al. 2019) (Jin et al. 2017) (Senouf et al. 2019) (Ulyanov et al. 2018).

Table 3.1 summarizes different physics-informed ML methods, along with the benefits of using such methods, such as increased ML model performance, interpretability, accuracy, and generalization to other data sets. It also indicates when they are generally applicable (usage); e.g., when there is a known physical equation, intermediate physical variables, previously trained weights, and whether a physics model, such as finite element model, is available.

For this project, ML algorithms were used to overcome challenges with traditional numerical methods, such as computational cost and potential alleviation of convergence issues. In this chapter, the feed-forward pass of an NN for a static problem is presented using an SDOF with a single stiffness value. The physics-informed NN methods discussed in this chapter include the physics-informed loss function and the hybrid physics NN model. To learn the solution to a static problem, weight initialization of the NN and data normalization can be very important and these were explored based on the NN performance. The best NN architecture was identified from different hyperparameters, and the results and conclusions are presented for the static problem at the end of the chapter.

3.2. Static Model Description

In this study, a static load, F, was applied to a linear-elastic spring with stiffness, k, to estimate the displacement, x, using an NN; the spring model is shown in figure 3.2. A linear-elastic model for static loading can be calculated by using Hooke's law:

$$F = kx \tag{3.1}$$

The NN inputs were the load, F, and stiffness k, and the output of the NN was the displacement, x, of the spring. A constant stiffness of $k = 1$ was initially selected to make the load in Equation 3.1 equal

Table 3.1: Different types of ML methods (Willard et al. 2020)

Physics-informed ML Method	Benefit	Usage
Loss Function	Improved accuracy, Improved generalization, Reduced number of iterations	Known physical equation
Architecture	Improved interpretability, Improved accuracy, Improved generalization	Hard constraints, or intermediate physical variables
Initialization	Reduced number of iterations Improved accuracy	Similar trained ML model available used for different purposes
Hybrid	Improved accuracy	Physics model already available

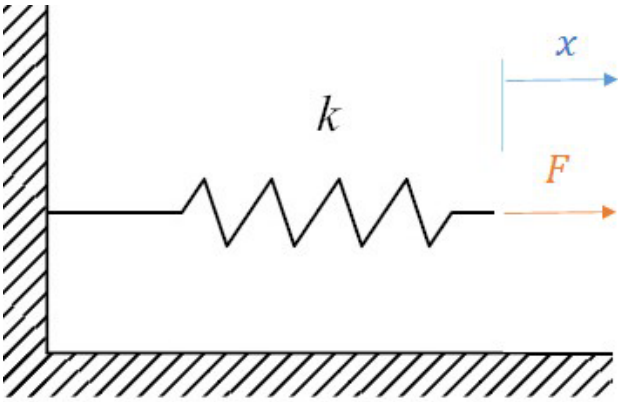

Figure 3.2: Linear-elastic spring

to the displacement, or $F = x$, to gain insight into how the NN performed with a simple example.

3.3. Neural Network Estimation

The NN is trained by updating the weights and bias to estimate the solution of a problem. For a single hidden layer with two neurons, as shown in figure 3.3, the feedforward NN without bias equation is:

$$a_1 = w^{(1)}_{11}F + w^{(1)}_{12}k$$
$$a_2 = w^{(1)}_{21}F + w^{(1)}_{22}k$$

For an activation function g(.):

$$a_1 = g(w_{11}^{(1)}F + w_{12}^{(1)}k)$$
$$a_2 = g(w_{21}^{(1)}F + w_{22}^{(1)}k)$$
$$\tilde{x} = w_{11}^{(2)}a_1 + w_{12}^{(2)}a_2$$

$$\tilde{x} = w_{11}^{(2)}g(w_{11}^{(1)}F + w_{12}^{(1)}k) + w_{12}^{(2)}g(w_{21}^{(1)}F + w_2^{(1)}k) \tag{3.2}$$

where $w_i^{(k)}$ are the weights and i refers to node in the connection layer, j refers to the node to the originating layer, and k refers to the number of layers. $\tilde{x}$ refers to the estimate from the NN.

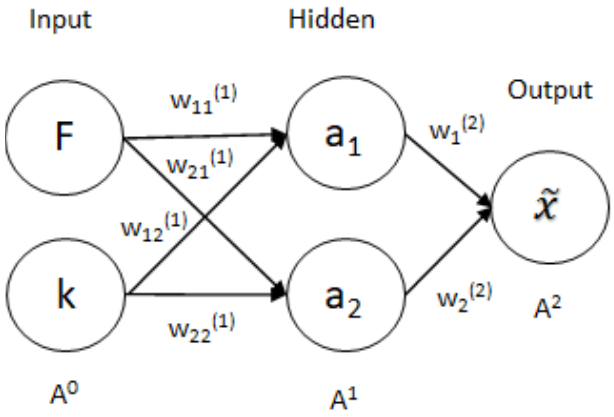

Figure 3.3: NN with single hidden layer, two neurons and no bias

Hooke's law in Equation 3.1 is of a linear form, but Equation 3.2 using the NN represents a nonlinear form. If the NN is used without a nonlinear, i.e., linear, activation function, then Equation 3.2 becomes

$$\tilde{x} = w_{11}^{(2)}w_{11}^{(1)}F + w_{11}^{(2)}w_{12}^{(1)}k + w_{12}^{(2)}w_{21}^{(1)}F + w_{12}^{(2)}w_2^{(1)}k \tag{3.3}$$

which simplifies to:

$$F(w_{11}^{(2)}w_{11}^{(1)} + w_{12}^{(2)}w_{21}^{(1)}) = \tilde{x} - k(w_{11}^{(2)}w_{12}^{(1)} + w_{12}^{(2)}w_2^{(1)}) \tag{3.4}$$

For the simple example, when Equation $k = 1$ in 3.1 and $F = x$, the load weight terms should add to unity, and the stiffness weight terms should add to zero.

3.4. Weight Initialization

To initiate the learning process, two different weight initializations were explored, including Xavier and He weight initializations. Weights and bias initialization are important for training an NN. Normally, weights and bias are initialized as small random values, but these random values can be problematic as the NN can get stuck in non-optimal minima, and the algorithm can fail to find the best solution for the problem (Goodfellow et al. 2016).

3.4.1. *Xavier Weight Initialization*

There are many different activation functions that can be used to introduce nonlinearity in the NN. These activation functions have unique properties, that can change the performance of the NN. The weight initialization introduced for sigmoid and tanh activation functions is known as Xavier or Glorot initialization (Glorot and Bengio 2010).

The Xavier weight initialization assumes a uniform distribution for random numbers between the range of $[-1/sqrt(n+m), 1/sqrt(n+m)]$, where n is the number of input neurons and m is the number of output neurons.

3.4.2. *He Weight Initialization*

Another type of weight initialization is the He initialization technique introduced by Kaiming He (He et al. 2015). In his study, He stated that there is no evidence of "clear superiority" of the He weight initialization over the Xavier initialization, but the He initialization is often used for ReLU activation function.

The He weight initialization assumes a uniform distribution having a mean of zero and a standard deviation of $sqrt(2/n+m)$.

3.5. Data Normalization

Several methods of normalizing the data were explored. If one input value is too big compared to other input values in the data, it will be assigned a higher weight value and dominate the results. The gradient descent algorithm is:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \alpha \frac{\partial C(w,b)}{\partial w_{ij}^{(l)}}$$

$$\frac{\partial C}{\partial w_{ij}^{(l)}} = h_j^{(l)} \delta_i^{(n_l)}$$

$$\delta_i^{(n^l)} = -(y_i - h_i^{(n_l)}) g'(z_i^{(n_l)})$$

which can be rewritten as:

$$w_i^{(l)} = w_i^{(l)} - \alpha[h_j^{(l)} - (y_i - h_i)g'(z_i^{(n_l)})] \tag{3.5}$$

The presence of $h_i^{(n^l)}$ in Equation 3.5 shows that the inputs affect the step size α in the gradient descent algorithm. Normalizing the inputs on the same scale helps the gradient descent algorithm to converge quickly.

In this study, Min-Max normalization was utilized. Min-Max normalization or feature scaling is a method to normalize the data so that the data set has a range between 1.0 and 2.0.

$$Inputs' = \frac{Inputs - min(Input)}{max(Input) - min(Input)} + 1 \tag{3.6}$$

For the $k = 1$ problem in Equation 3.1, the following normalization techniques were studied:

1. Normalizing the load, F, only by Min-Max normalization.
2. Normalizing the load, F, and displacement, x, by Min-Max normalization

3.6. Physics Learning NN

The NN was trained to satisfy Hooke's law by using the normalized data from Equation 3.6. The physical equation in Equation 3.1 was incorporated into the learning by adding the residual calculated from OpenSees into the NN loss function. The inputs (F and k) are given to the NN to estimate the output (x), which is then given to OpenSees to calculate the unbalance force and residual. The residual is added to the loss function and then backpropagated upon to update the weights of the model (figure 3.4).

1. **Soft constraints**: The loss function is minimized to improve the quality of the NN rather than obtain a certain precision. There are two limitations to loss functions with soft constraints. First, there is no guarantee that the constraints will be satisfied. Second, it is necessary to choose the loss terms in the loss function wisely according to their relative importance to the problem (Márquez-Neila et al. 2017).
2. **Hard constraints**: The loss function is minimized to be near-equal to the machine precision if the solution to the problem exists (Márquez-Neila et al. 2017). There are limitations of having loss functions with hard constraints as the NN often needs to learn millions of free parameters. This can overwhelm the optimization problem (Pathak et al. 2015).

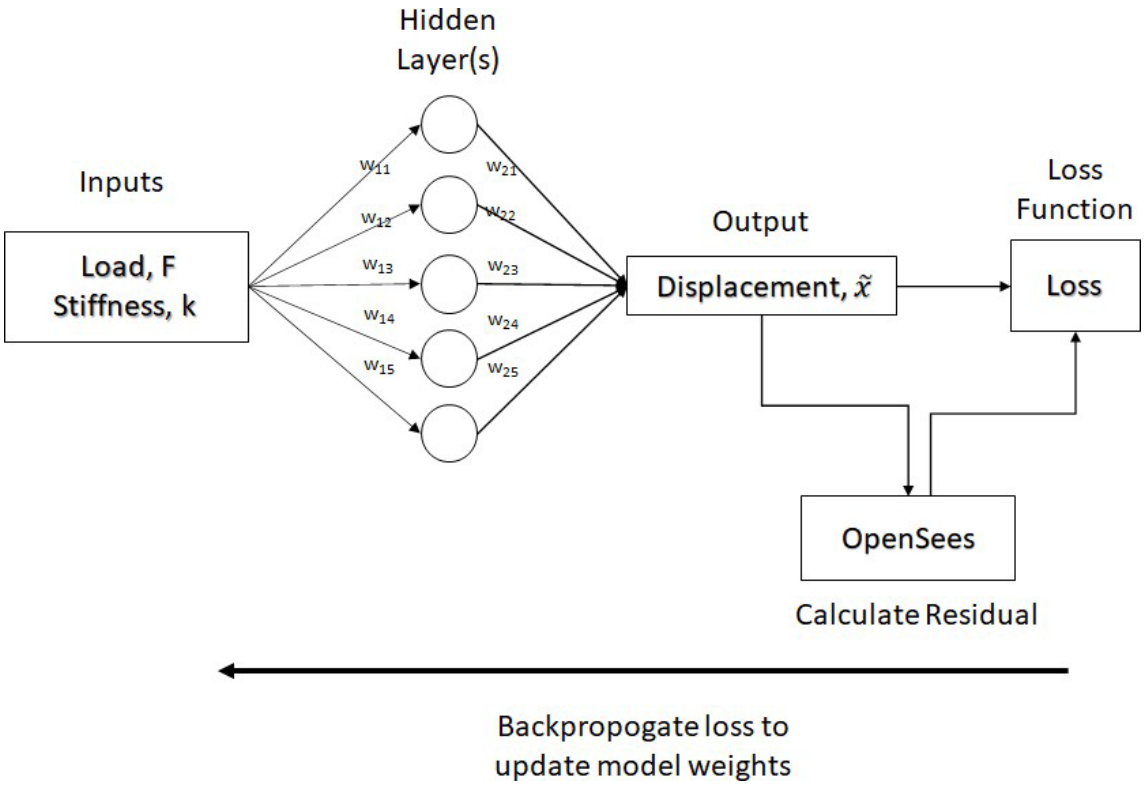

Figure 3.4: Physics neural network

3. **Regularizer**: The residual can be added as a penalty to the loss function. This makes the physics term a regularizer which is minimized by the NN.

Herein, physics was embedded into the loss function during the training process by using soft constraints with either the data or the residual in the loss function. Two different types of loss functions were utilized:

- Residual loss R, using the unbalance force vector, P_u, for Equation 3.1

$$\mathrm{P_u} = \mathrm{F} - \mathrm{k}\tilde{x} \tag{3.7}$$

$$Loss = L_R = (\mathrm{P_u})^{\mathrm{T}}(\mathrm{P_u}) \tag{3.8}$$

- Energy loss, using the unbalance force vector and differences between the output displacements from ML, x, and the training data displacements from OpenSees, $\tilde{x}$, defined as:

$$Loss = L_E = |(\mathbf{x} - \mathrm{x})^{\mathrm{T}}(\mathrm{P_u})| \tag{3.9}$$

Unlike residual loss, energy loss minimizes the error between both the displacements and unbalance forces.

3.7. Neural Network Architecture

Several NN architectures were explored to determine the best number of neurons, hidden layers, and activation functions. A good NN architecture design is necessary to have good performance. As discussed in Chapter 2, an NN becomes more flexible when more hidden layers are added to the network, but too many hidden layers can cause the NN to overfit the data. Once an NN architecture was selected, loss functions including the residual were utilized.

3.7.1. *One Layer without Bias Neural Network*

One hidden layer NN without bias was modeled with the normalization techniques for $k = 1$ and without the physics loss functions. Figure 3.3 shows a NN with a single hidden layer, two neurons, and no bias with a learning rate of 1e-3 and an Adam optimizer. Table 3.2 shows the performance of the NN for different normalization techniques. Training was discontinued after a maximum number of 50,000 epochs. Not all NNs converged within the 50,000 epochs. Herein, convergence was defined by a tolerance of 1e-12.

Table 3.2: Neural network with single hidden layer and no bias

Activation function	Data normalization type	MSE test error	Iterations
tanh	Normalizing Load	1.206e10	50000
ReLU	Normalizing Load	1.137e-21	1380
Linear	Normalizing Load	1.450e-21	983
tanh	Normalizing Load + Displacement	1.341e10	50000
ReLU	Normalizing Load + Displacement	8.745e-08	16
Linear	Normalizing Load + Displacement	2.107e-12	15

The normalization techniques converged with the ReLU and linear activation function. The tanh activation function did not converge, because of the nonlinearity introduced into Equation 3.2. Figure 2.6 in Chapter 2 also shows that tanh is not suitable for a linear function such as $F = x$.

3.7.2. *Two Layer without Bias Neural Network*

A two hidden layer NN without bias was modeled for each normalization techniques without physics. The two hidden layers had two neurons and no bias, with a learning rate of 1e-3 and an Adam optimizer. Table 3.3 shows the performance of the NN for different normalization techniques

Table 3.3: Neural network with two hidden layers and no bias

Activation function	Data normalization type	MSE test error	Iterations
tanh	Normalizing Load	1.206e10	50000
ReLU	Normalizing Load	5.902e-20	188
Linear	Normalizing Load	2.517e-20	152
tanh	Normalizing Load + Displacement	4094	50000
ReLU	Normalizing Load + Displacement	6.957e-10	14
Linear	Normalizing Load + Displacement	1.424e-11	13

The two layers without bias became more flexible and fewer iterations were needed for convergence compared to the one layer case. The results for the normalization type followed a similar trend similar to that of the one layer case, with ReLU and linear activation functions outperforming tanh.

3.7.3. *Two Layer with Bias Neural Network*

Figure 3.5 shows two hidden layers with bias with two neurons per layer. Table 3.4 and Table 3.5 show the results for a two hidden layer NN with bias and two neurons, with and without the residual. Without and with physics, non-convergence was considered at 50,000 and 5,000 epochs, respectively. The learning rate was 1e-3 with an Adam optimizer.

Table 3.4: Neural network with two hidden layers and bias

Activation function	Data normalization type	MSE test error	Iterations
tanh	Normalizing Load	1.206e10	50000
ReLU	Normalizing Load	1.947e-21	84
Linear	Normalizing Load	8.799e-21	56
tanh	Normalizing Load + Displacement	6158	50000
ReLU	Normalizing Load + Displacement	3.728e-20	7
Linear	Normalizing Load + Displacement	1.889e-19	4

Of the architectures studied, two layers with bias resulted in the least number of iterations for convergence. ReLU and linear outperformed tanh in every NN architecture for k = 1, for all the normalization techniques with and without physics. The residual and energy loss functions both took the same number of epochs to converge for the ReLU and tanh activation functions. The normalization technique, which included normalizing the load and displacement, did not perform well with the energy loss, as the unbalance force and the error had different units. For the data normalized case, the unbalance force was in the original domain (not normalized) for OpenSees to analyze the results, whereas displacement values were

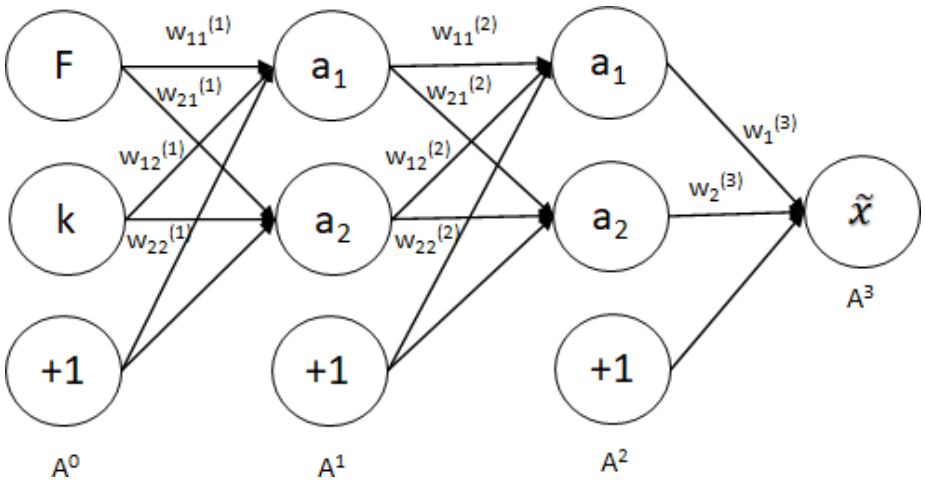

Figure 3.5: NN with two hidden layers, two neurons, and bias

Table 3.5: Neural network with two hidden layers and bias with physics loss function

Physics loss function	Activation function	Data normalization type	MSE test error	Iterations
L_R	tanh	Normalizing Load	2.756e9	5000
L_R	ReLU	Normalizing Load	8.653e-22	85
L_R	Linear	Normalizing Load	4.344e-21	56
L_E	tanh	Normalizing Load	2.756e9	5000
L_E	ReLU	Normalizing Load	1.890e-21	85
L_E	Linear	Normalizing Load	4.344e-21	56
L_R	tanh	Normalizing Load + Displacement	2.756e9	5000
L_R	ReLU	Normalizing Load + Displacement	2.094e-21	85
L_R	Linear	Normalizing Load + Displacement	4.344e-21	56
L_E	tanh	Normalizing Load + Displacement	6.937e9	5000
L_E	ReLU	Normalizing Load + Displacement	6.937e9	5000
L_E	Linear	Normalizing Load + Displacement	6.937e9	5000

normalized. The residual loss function did not have this problem because it involved only the unbalance force from OpenSees.

3.8. Hyperparameter Tuning

The effects of batch size and the number of neurons in the hidden layers were studied to select the best-trained model. The learning rate was 1e-3 with the activation function ReLU with load normalization. Figure 3.6 shows that increasing the number of neurons in the first hidden layer from 2 to 10 resulted in a

decreasing number of iterations. For one layer, the decrease in the number of iterations was evident, but for the two layers with and without bias the number of iterations decreased only slightly. The flexibility of the network reduced the needed number of iterations with changes in the NN architecture. The number of iterations was also reduced with an increasing number of neurons in the second hidden layer (figure 3.7), while the first layer hidden neurons were kept constant (two neurons).

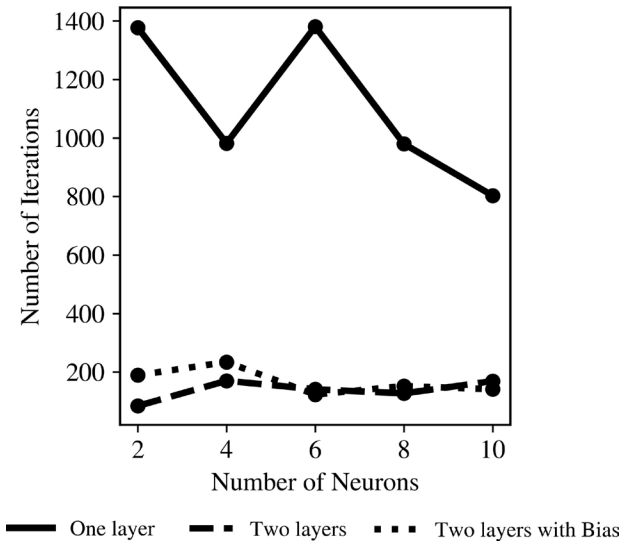

Figure 3.6: Number of neurons in first hidden layer vs number of iterations

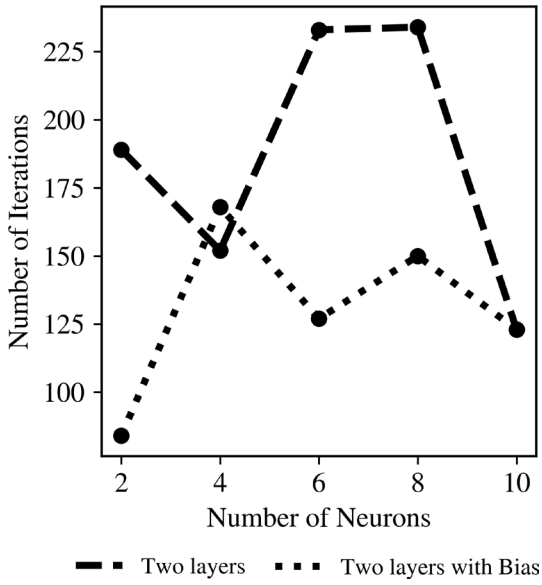

Figure 3.7: Number of neurons in second hidden layer vs number of iterations

Figure 3.8 shows the effect of batch size on the number of iterations for different NN architectures. Training the NN with a small batch size made the NN converge faster, as there were more updates to the

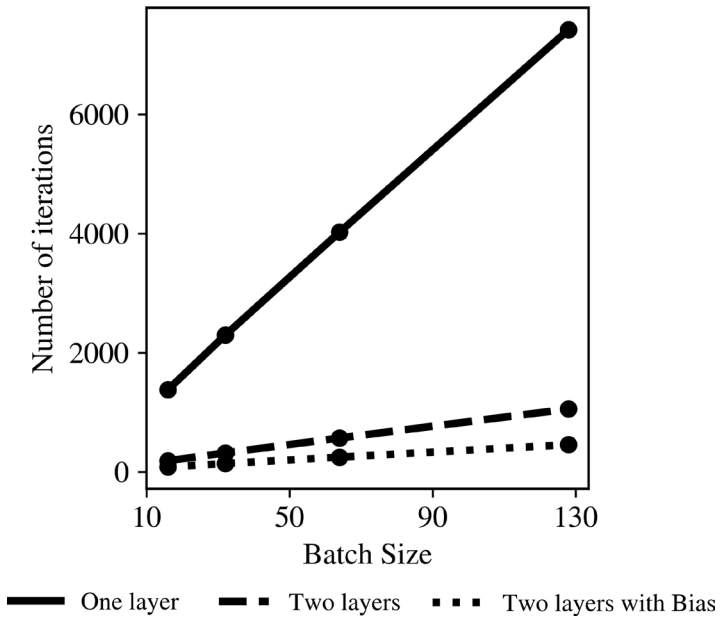

Figure 3.8: Batch size vs number of iterations

NN and the weights updated more frequently.

3.9. Results

The neural network architecture with two hidden layers with bias having two neurons in each layer had the best performance with a static SDOF and $k = 1$. The normalization technique resulting in the smallest number of iterations was normalizing load, F, and displacement, x, by the Min-Max normalization. The activation function that performed the best was the linear function, which best fit Hooke's Law, followed by ReLU. Figure 3.9 shows the results of the actual value and the estimated predicted value from the NN.

The results were linear when $k = 1$, and the NN well estimated the values from OpenSees. These results were trained with a learning rate of 1e-3 and a batch size of 16 with the Adam optimizer.

The convergence of the NN was smooth (Figure 3.10) for both normalization types. The NN with the gradient descent algorithm and a learning rate of 1e-3 resulted in convergence issues. The Adam algorithm was found to be more stable than the gradient descent algorithm for this problem. A very small learning rate of 1e-8 was needed for the gradient descent algorithm to reach convergence.

Regardless of whether the loss function used physics, the NNs performed similarly (table 3.4). The residual and energy loss functions converged for the same number of iterations as that without physics for the ReLU and linear activation function. The physics-informed loss functions did not have a significant effect on convergence because the dataset was linear, and the NN could easily identify patterns with and without physics.

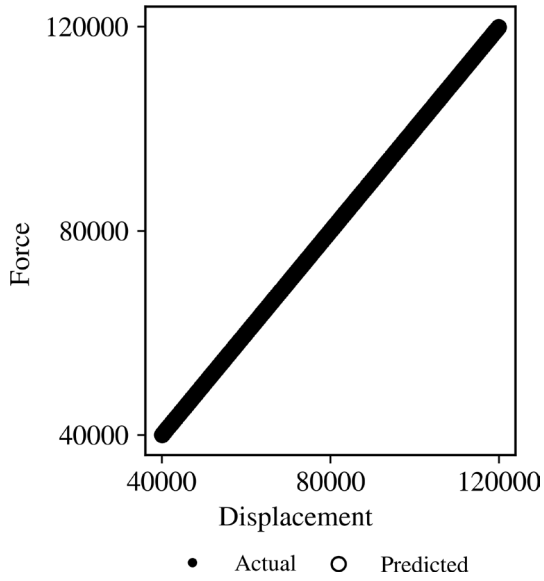

Figure 3.9: Results of the NN for static problem when $k = 1$

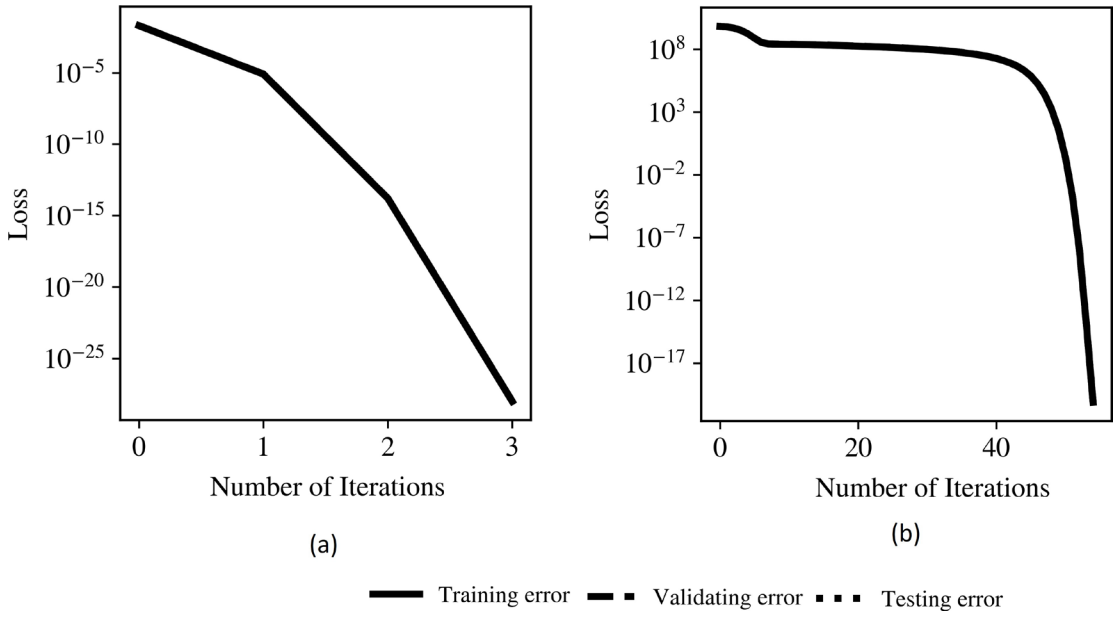

Figure 3.10: MSE error vs number of iterations. (a) Normalizing load and displacement, (b) Normalizing load

3.10. Extrapolation

The performance of the NN was checked by extrapolating the dataset and using the NN model to estimate values far away from the original training dataset. The ReLU activation function performed in a

manner similar to that of the linear activation function, as shown in table 3.2, table 3.3, and table 3.4, but it failed to estimate negative values of displacement (figure 3.11). The ReLU activation function does not estimate negative values because it is a piecewise linear activation function that is zero when the output is negative and linear when the output is positive. In contrast, the linear activation function estimates positive and negative values (figure 3.12), as it best matches Hooke's law.

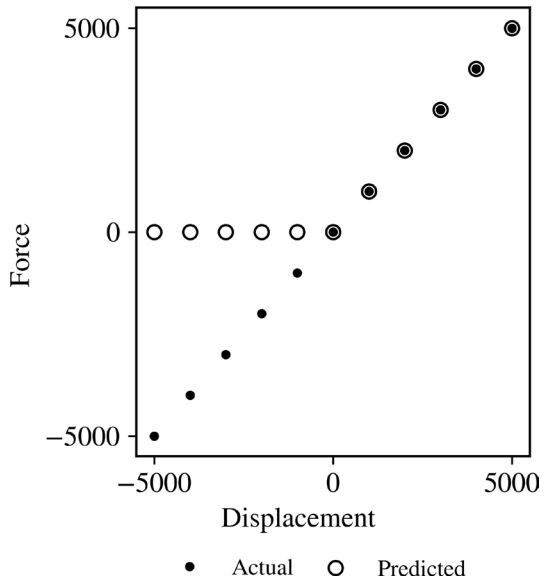

Figure 3.11: ReLU extrapolation

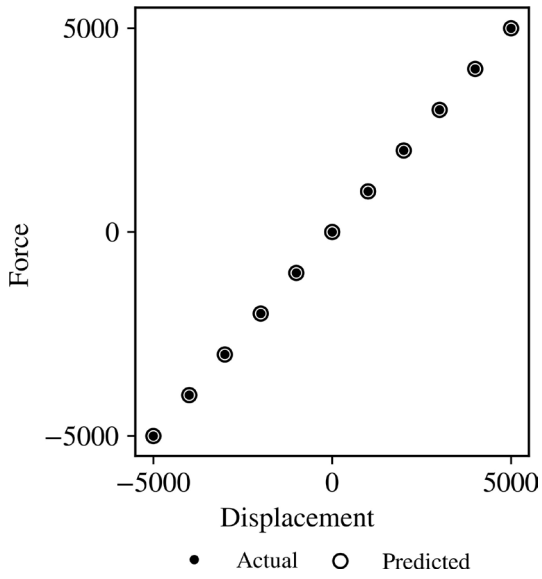

Figure 3.12: Linear extrapolation

3.11. Conclusions

An NN was trained by using an elastic SDOF with a load, F, and a stiffness, k, for displacement, x. The stiffness of the model was assumed to be constant. Physics was included in the loss function of the NN as a soft constraint and backpropagated to update the loss function. Several types of normalization techniques were used to assess the performance of the NN. Different types of NN architectures with different numbers of neurons and activation functions were studied to select the one most suited to this problem.

The performance of the NN was best when the ReLU and linear activation functions were used because Hooke's law is linear. The ReLU and linear activation functions performed similarly when the testing set had positive values, but ReLU failed to perform when the testing set had negative values because ReLU returns zero on the negative branch. Tanh, a nonlinear activation function, did not converge due to the linear nature of the dataset.

Normalizing the inputs and the outputs sped up the NN, with fewer iterations needed for convergence. Physics was introduced into the NN by embedding a residual extracted from OpenSees into the loss function. The performance of the NN changed a little by incorporating physics into the NN. Both the physics residual and energy loss functions performed similarly with $k = 1$.

This study was kept relatively simple to select an appropriate NN architecture. The structural model was for an SDOF with constant stiffness $k = 1$ and a linear force-displacement relationship. The next chapter investigates the use of an NN with variable stiffness with and without physics, which can be extended for both single- and multiple-degree-of-freedom-systems and nonlinear response in the future.

4. STATIC MODEL WITH VARYING STIFFNESS

4.1. Introduction

Neural networks (NN) are data-driven in nature, as they rely only on the available dataset values during training to update the weights of the NN model. The NN maps the inputs to the outputs, calculates the loss, and then updates the weights of the network to reduce the loss function. Thus, the underlying physical equations can be disregarded by the NN when a solution for the problem is estimated. Since the NN does not use scientific theory, it can be limited to performing well only within the range of the training dataset. Therefore, a physics-informed NN model that uses both scientific theory and is based on data-driven concepts can be advantageous (figure 3.1).

In this project, the loss function of the NN was modified by adding the residual, which penalizes the NN if the governing equations are not satisfied. Embedding the residual into the loss function helps in minimizing overfitting of the data and makes the NN more reliable, and it can help in reducing the number of iterations for convergence during training (Willard et al. 2020).

This chapter extends the previous chapter by using a static model with variable stiffness. The feedforward pass with the variable stiffness problem was compared with the physical equations for the static model, and it was inferred that the NN could not learn the physical equation in its original form. The form of the physical equation was changed by applying log normalization to the data and different loss function performance was evaluated in a manner similar to that described in the previous chapter.

4.2. Static Model Description

As discussed in Chapter 3, Hooke's law was learned by using an SDOF. Work described in Chapter 3 was limited to cases of a single stiffness value. Here, multiple stiffness values were used during the training so that the learned model could better generalize to different structures with different stiffnesses.

Different static models with random stiffness values were subjected to random force values to form a training set for the NN. The NN inputs included the stiffness and force values. The output of the NN was the displacement for these different force-stiffness conditions. The model was designed as a zero-length element in OpenSees.

4.3. Neural Network Feedforward Calculation

The NN estimated displacements by comparing the predicted values output by the NN to the training values extracted from OpenSees, characterizing the loss. The NN minimizes loss by updating the

weights and bias to minimize the error between the training values and estimates from the NN. The larger the loss, the larger error from the machine learning model.

To calculate the loss, estimates for displacements were calculated from the inputs using a feedforward pass.

For a single layer NN with two neurons (figure 3.3), the feedforward pass calculation for two inputs (k and F) to get an estimate of the output (x) was like that defined in Chapter 3.

To minimize the loss and make the estimates of the NN as close to the original value as possible, the NN had to learn Hooke's Law. Rearranging Equation 3.1 for the displacements, x:

$$x = \frac{F}{k} \tag{4.1}$$

A comparison of Equation 4.1 and Equation 3.4, shows that the estimates from the NN can never be equal no matter which weight values are used. Therefore, Equation 4.1 and Equation 3.4 were reformed with log normalization to better represent Hooke's Law and reduce the loss of the NN.

4.4. Log Normalization

To align the form of the NN to be similar to Hooke's Law, log normalization was applied to Equation 4.1, and the log-normalized values were used in Equation 3.4. Equation 4.1 then became:

$$\ln(x) = \ln(F) - \ln(k) \tag{4.2}$$

and Equation 3.4 became:

$$\ln(\mathrm{x}) = w_{11}^{(2)} g(w_{11}^{(1)} \ln(\mathrm{F}) + w_{12}^{(1)} \ln(\mathrm{k})) + w_{12}^{(2)} g(w_{21}^{(1)} \ln(\mathrm{F}) + w_{2}^{(1)} \ln(\mathrm{k})) \tag{4.3}$$

With this normalization, x and x could be equal to each other, given some combination of weights for the loading and stiffness terms in Equation 3.4.

4.5. Physics-Informed NN

The NN was trained by using the log-normalized data in Equation 4.2. Figure 4.1 shows a physics-informed NN, wherein in the training phase, the inputs (F and k) were mapped to the output/estimates (x) to

calculate the loss. The estimates of the NN were used to calculate the unbalance force and residual using OpenSees and they were added to the loss function. The loss function, which included the loss from the data and the residual, was then back propagated to update the weights of the physics-informed NN. In this fashion, the physical equations were embedded into the learning process. The unbalance force (P_u) and residual (L_R) were like those used in Chapter 3.

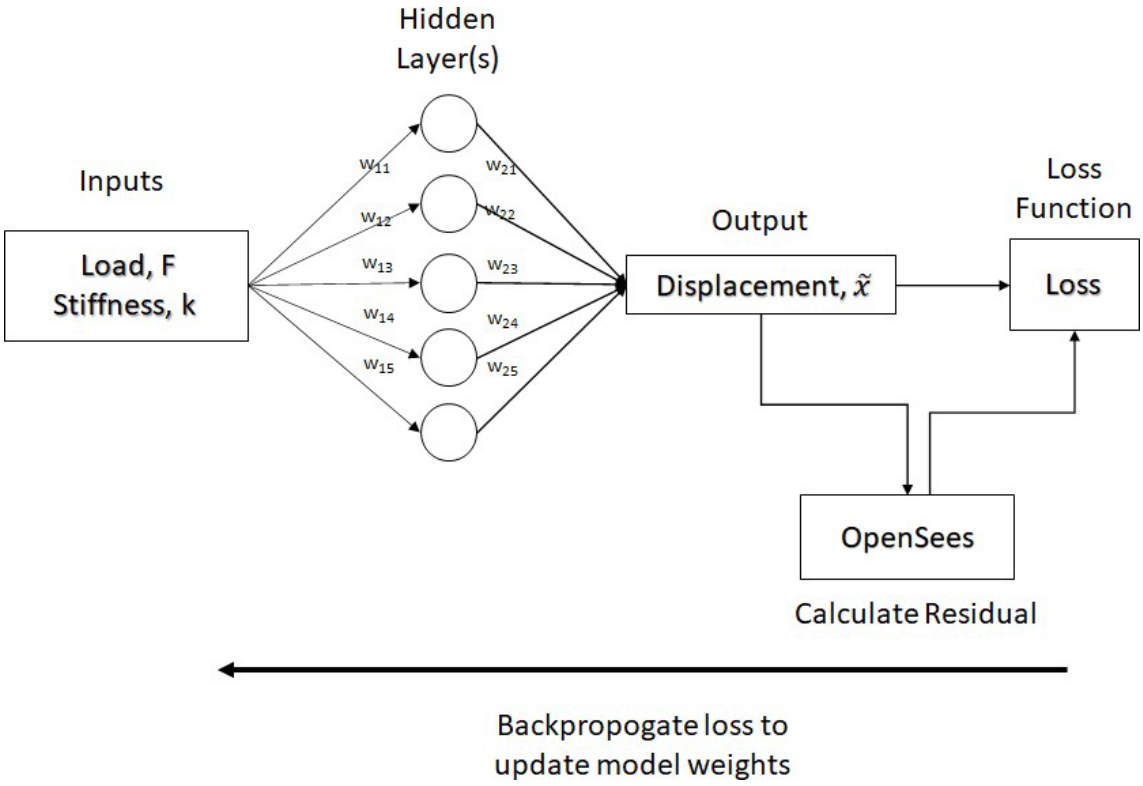

Figure 4.1: Physics neural network

The residual was added to the loss function as a regularizer term so that it could be used in combination with the data-driven loss. A parameter λ was used for the "physics" part in the loss function to decide how much the NN should learn from the physics of the problem versus the data. This allowed for control of the NN adoption between the physics and data-driven part of the problem. Equation 4.4 shows the regularization constraint for a static problem.

$$loss = L_D + \lambda L_R \tag{4.4}$$

And,

$$L_D = \frac{1}{n}\sum_{i=1}^{n}(x - \hat{x})^2 \tag{4.5}$$

where $\hat{x}$ are the estimates from the NN, x are the original outputs, and L_R is the physics residual from OpenSees.

The λ term in the regularizer is an unknown value and varies based on the relative importance of the physical equations versus the training data for the problem. Many physics-informed neural networks have used Equation 4.4 to improve the accuracy of the NN estimates (Raissi, Z. Wang, et al. 2019) (Y. Yang and Perdikaris 2019) (Kharazmi et al. 2019), but the effects of λ are problem dependent. An improper selection of the λ value can lead to potentially unstable and unreliable results (Raissi, Yazdani, et al. 2018) (Sun et al. 2020).

This study explored the performance of soft constraints and regularization of the NN. The soft constraint term include the data-driven or residual in the loss but not both, whereas the regularization constraint contains both the data-driven and physics of the problem.

4.6. Regularization Loss Function with Physics

Equation 4.4 uses a penalty (λ) on the residual term in addition to the data-driven loss. Using the residual as a regularizer in the loss function can be interpreted as weighted training in combination of learning the best fit of the data (data-driven).

One approach has used a varying λ that was updated by the NN after each iteration (S. Wang et al. 2021). The λ value is a function of the weights of the NN, which are updated after each iteration. This approach optimizes the value of λ just as the NN weights:

$$\lambda = \frac{Max(\boldsymbol{w})}{Avg(\boldsymbol{w})} \tag{4.6}$$

Table 4.1 shows different λ values tested against the performance of the NN for 5,000 epochs, an Adam optimizer, and a learning rate of 1e-03. The convergence criterion was 1e-12. The linear activation function converged for all λ values except for 1e-09. The ReLU activation function did not converge but still has a small test loss.

4.7. Results

A NN with two hidden layers and bias was trained for the static model. The training dataset was

normalized by using the aforementioned Log normalization technique. The NN was trained with and

Table 4.1: NN results for different λ values

λ	Activation Function	MSE test error	Epochs
1e-09	ReLU	1.88E+27	5000
	Linear	4.71e+28	5000
1e-06	ReLU	4.22E-13	5000
	Linear	1.47e-21	1301
1e-03	ReLU	4.24e-24	5000
	Linear	4.24e-24	1613
1	ReLU	1.51E-05	5000
	Linear	1.49e-27	1743
1e+03	ReLU	4.11E-06	5000
	Linear	4.94e-29	2006
Varying λ	ReLU	4.63E-07	5000
	Linear	1.39e-13	1669

without the activation function $g(.)$ in Equation 4.3; i.e., the without case was equivalent to using a linear activation function. The NN results were trained with a learning rate of 1e-3 with a batch size of 16 and using an Adam optimizer.

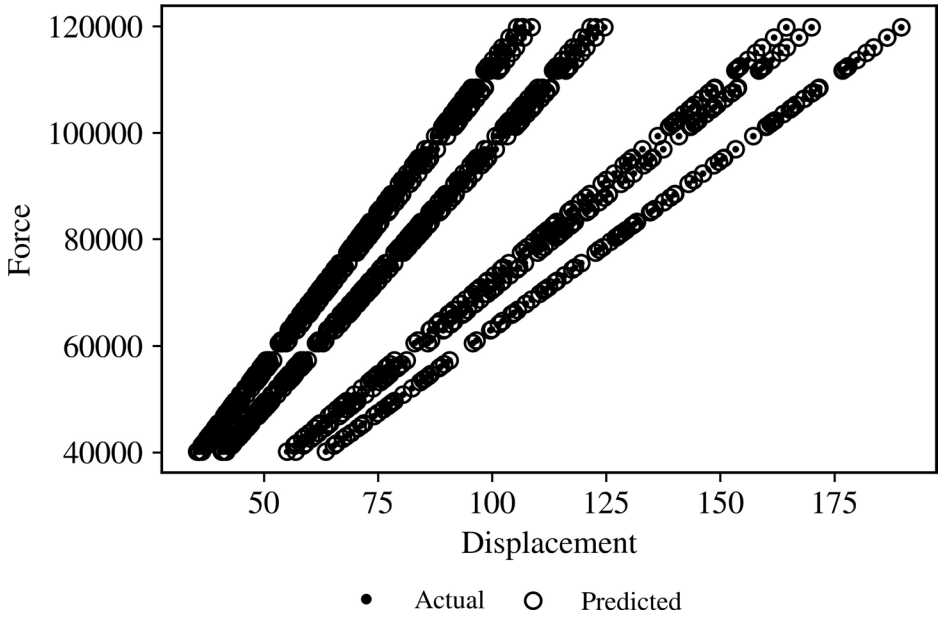

Figure 4.2: Test results of the NN for varying stiffness of SDOF system

Figure 4.2 shows results of the static model from OpenSees, which were near those estimated by the NN. Figure 4.3 plots different loss functions with different activation functions against the number of iterations (epochs). Most of the loss functions did not converge after 5,000 epochs. However, even without convergence, figure 4.4 shows that they performed well on the test set, and the loss was small.

All loss functions performed well when the NN was mapping the inputs to the outputs linearly using the log normalization. When the ReLU activation function was used and nonlinearity was introduced into the NN, the results did not converge but produced a small loss value after 5,000 epochs. The L_{data} and L_E functions with linear (or no activation function) converged, as can be seen in figure 4.3. The performance of the NN did not change when the residual was introduced into the loss function, as the dataset could be easily fit with the log normalization form.

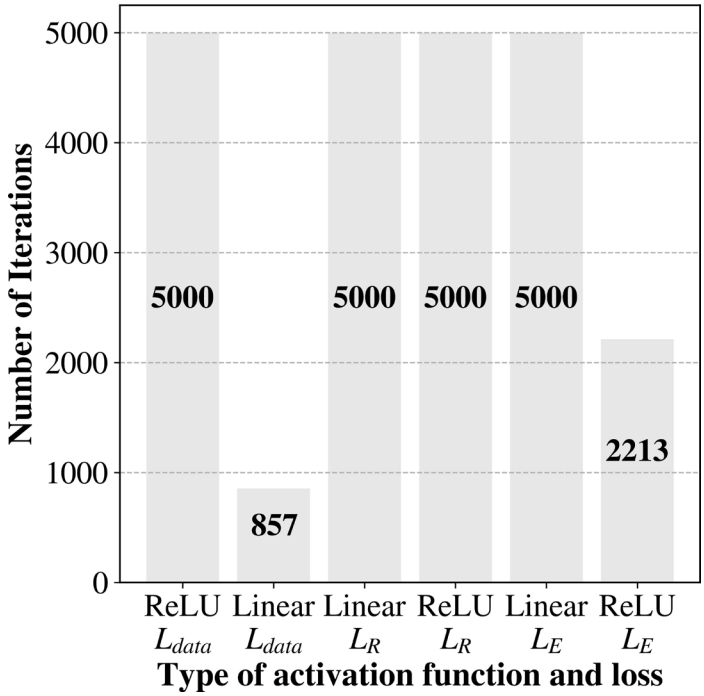

Figure 4.3: Different types of loss functions vs number of iterations

4.8. Conclusions

An NN was trained for an elastic SDOF model with variable stiffness. Physics was included in the loss functions of the NN, and different loss function performances were compared. A log normalization technique was applied to the NN feedforward pass so that the NN could learn the linear form of the underlying physical equation of the static model. The NN was able to estimate the values from OpenSees and performed well on the extrapolated values (figure 4.5).

The data-driven NN and physics-informed NN had similar performance for the static model. Adding physics to the loss function did not affect the results because this was a simple model and the results were linear. The NN with a linear activation function performed slightly better than the NN with nonlinear activation functions because the model was linear-elastic.

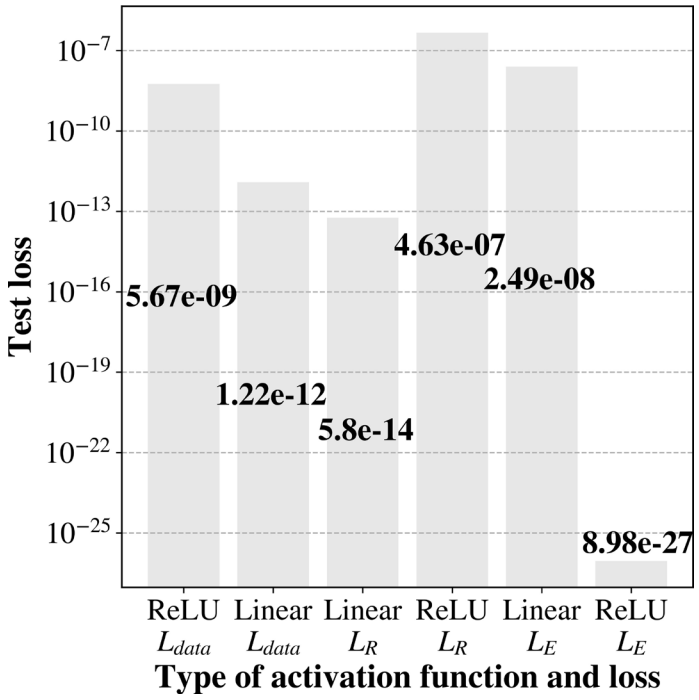

Figure 4.4: Different types of loss functions vs test loss

Multiple degree of freedom systems (MDOF) should be studied in the future to evaluate the performance of the physics-informed loss function. The effects of nonlinearity on the performance of the NN should also be studied. Having a more complex problem, such as dynamic analysis, can be used to further evaluate the difference in performance between data-driven and physics-informed loss functions. This is explored in the next chapter.

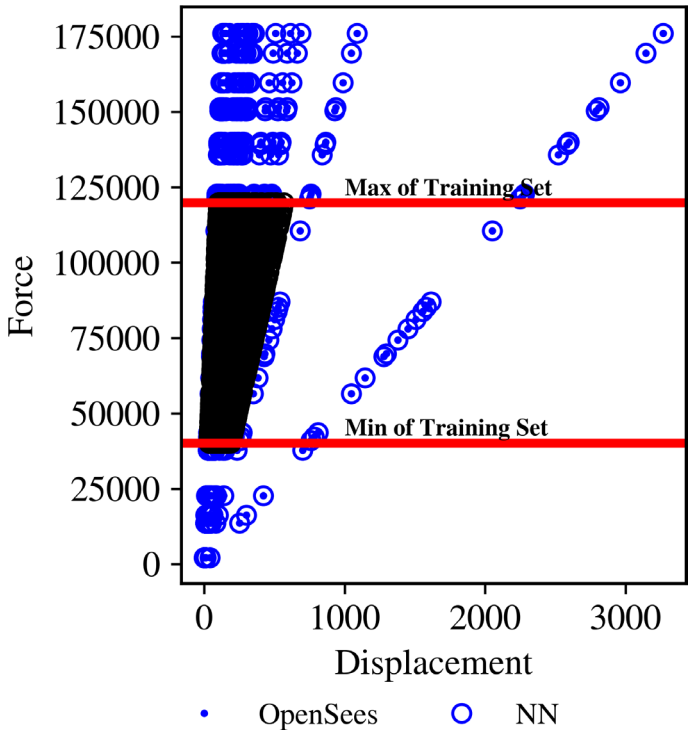

Figure 4.5: Extrapolation results with training

5. DYNAMIC MODEL

5.1. Introduction

Machine learning (ML) and deep learning (DL) algorithms have been used in structural engineering applications (Thai 2022), such as estimating the load-bearing capacity of isolated structures, the mechanical properties of concrete, structural health monitoring, the fire resistance of structures, and analysis and design of structures.

Many ML and DL algorithms do not use scientific theory when estimating the results. Thus, the results of ML and DL algorithms are purely data-driven, which can be a problem in terms of robustness of the model and interpretation. In this study, the equations of motion were included into the ML and DL algorithms in an effort to make the results more reliable. This chapter extends the previous two chapters by descripting the use of an SDOF oscillator subjected to free vibration using an initial displacement value.

5.2. Dynamic Model Description

A dynamics model consisting of a spring with a mass, m, and stiffness, k, was initially displaced with a displacement, x, and then the free vibration of the system was recorded. The free vibration of a dynamic system with mass, m, and stiffness, k in time, t, is given as:

$$m\ddot{x} + kx = 0 \tag{5.1}$$

Dividing by the mass,

$$\ddot{x} + \omega^2 x = 0 \tag{5.2}$$

where $\ddot{x}$ is the acceleration of a single degree of freedom oscillator at time t, x is the displacement of a single degree of freedom oscillator at time t, and ω is the circular frequency of a single degree of freedom oscillator.

A single degree of freedom (SDOF) dynamic problem with $m = 1$ and $k = 1$ was subjected to random initial displacements. The displacement values were recorded for time t seconds. The NN inputs were the acceleration at time t, circular frequency and displacement of the dynamic system at time t. The output of the NN was the displacement of the dynamic system at the next time step, $t + 1$.

5.3. Recurrent Neural Network

A recurrent neural network (RNN) is a deep learning algorithm that makes predictions using sequential data. A simple neural network or artificial neural network (ANN) is trained by treating each data point as an individual input even when the data points are related to each other. An RNN takes advantage of the correlation between data points and trains by using the prior information of the data points. RNNs are mostly used for time-series forecasting (Coulibaly and Baldwin 2005) and natural language processing (Yin et al. 2017).

5.4. RNN Architectures

RNN uses sequence information to improve the outputs. The architecture of an RNN is like an ANN with respect to having input layers, hidden layers, and output layers. The only difference is that an RNN keeps the output layer information for the next input layer. Figure 5.1 shows an ANN architecture on the left and RNN architecture on the right.

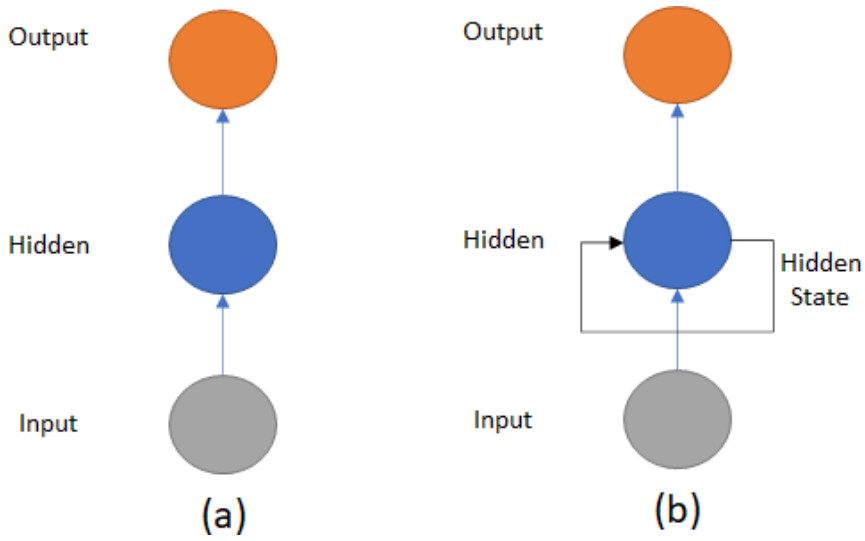

Figure 5.1: a) ANN architecture, b) RNN architecture

The RNN architecture can be varied by changing the number of inputs and outputs. The variations in the type of RNN architecture depend on the type of RNN needed to solve the problem. Figure 5.2 shows several architectures; one-to-one NN architecture is an ANN whereas other variations show RNN architectures.

The many-to-one is a recurrent architecture in which many inputs are given to the NN, and it assigns weights to all the inputs and learns from the combination of the many inputs to predict one output. The one-to-many RNN architecture has many outputs, which are combined to learn from a single input.

Another type of RNN architecture is many-to-many, in which many inputs are combined to predict many outputs all at once.

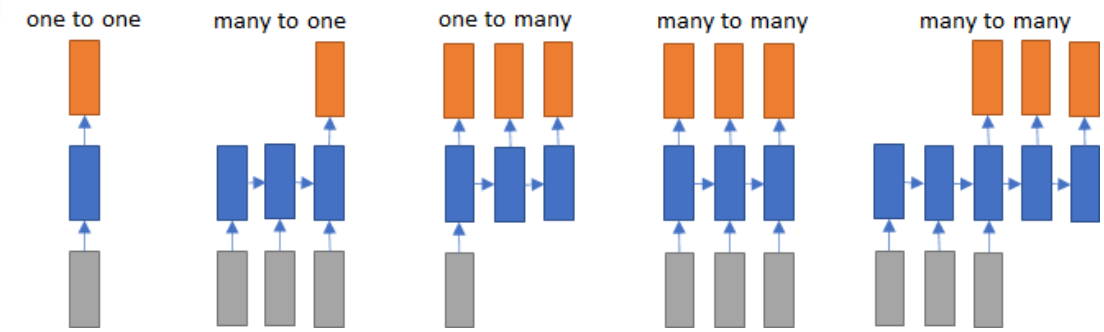

Figure 5.2: Variation in neural network architectures with different number of inputs and outputs

5.5. RNN Forward Pass

The RNN forward pass is like the forward pass of an ANN, the only difference being that the previous output information is used in the next input. The forward pass for an RNN architecture, shown in figure 5.3 with input I, hidden state h and output O from time step $t-2$ to time step $t+1$, is calculated at each time step as:

$$h_t = g_1(W_h h_{t-1} + W_i I_t) \quad (5.3)$$

$$O_t = g_2(W_o h_t) \quad (5.4)$$

Where W_i, W_h and W_o are the weights to the input, hidden, and output layer respectively; $g_1(.)$ and $g_2(.)$ are the activation functions; and h_{t-1} are the previous hidden state.

5.6. Data Normalization

An NN is sensitive to the information given to it as input. This is mainly due to how the NN estimates the values from the inputs through the forward pass equation. The step size of the gradient descent algorithm is influenced by the inputs to the NN. Therefore, it is essential to find a good normalization technique that best represents the features of the raw data efficiently. Many data normalization techniques have been discussed in (D. Singh and B. Singh 2020), which are described below.

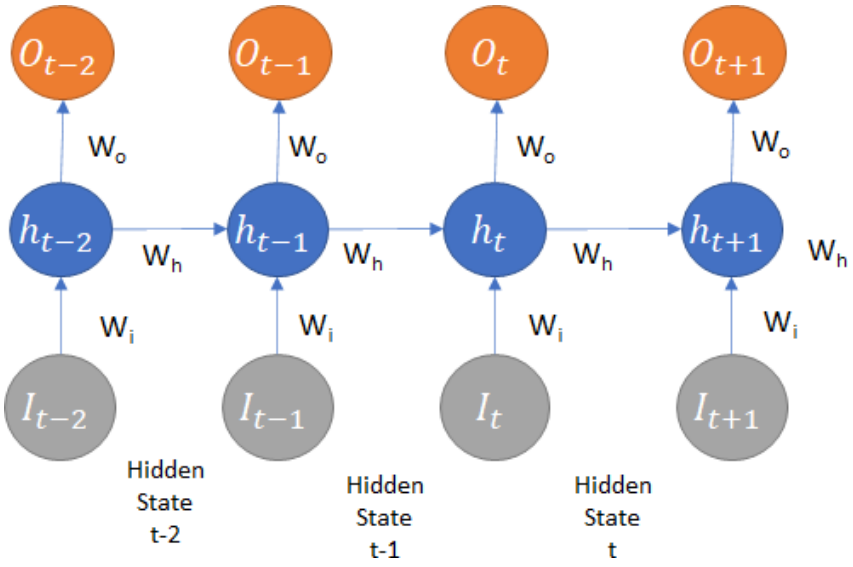

Figure 5.3: RNN forward pass

5.6.1. *Min-Max Normalization*

The Min-Max normalization technique, also known as feature scaling (FS), normalizes the input values by scaling the inputs to a desired range. In this study, the input values were normalized between 1 and 2, as FS used with some other function that involves the log, and the normal 0 to 1 value would not work. The min-max normalization is:

$$Inputs' = \frac{Inputs - min(Input)}{max(Input) - min(Input)} + 1 \tag{5.5}$$

5.6.2. *Variable Stability Scaling (VSS)*

The variable stability scaling method is similar to scaling the inputs to a standard normal distribution. The difference is that VSS multiplies the standard normal distribution by the mean and divides it by the standard deviation of the data, also known as coefficient of variation. This normalization causes the input values with a large standard deviation to have less importance and values with a smaller standard deviation to have higher importance.

$$Inputs' = \frac{Inputs - \mu}{\sigma}\,\mu \tag{5.6}$$

σ

5.6.3. *Pareto Scaling (PS)*

The Pareto scaling method uses the standard normal distribution, but the new inputs now have a variance equal to the standard deviation. PS minimizes the impact of noise and improves the representation of lower concentrated values.

$$Inputs' = \frac{Inputs - \mu}{\sqrt{\sigma}} \tag{5.7}$$

5.6.4. *Power Transformation (PT)*

The power transformation normalization transforms the data into having a variance that is constant across the dataset such that the dependent variable is equal across all values of the independent variable (homoscedasticity). PT is:

$$[illegible] = Inputs - min(Input) \tag{5.8}$$

$$Inputs = p - \mu_p \; where \; p = [illegible] \tag{5.9}$$

5.6.5. *Hyperbolic Tangent Normalization (TN)*

The hyperbolic tangent normalization technique was proposed by (Hampel et al. 2011). TN is not sensitive to outliers.

$$Inputs' = \frac{1}{2}(tanh(0.01(\frac{Inputs - \mu}{\sigma})) + 1) \tag{5.10}$$

5.6.6. *Sigmoidal Normalization Logistic Sigmoid (LS)*

The logistic sigmoid (LS) normalization is based on the activation function sigmoid of the NN. LS is a nonlinear transformation of the inputs to reduce the effects of the outliers.

$$Inputs' = \frac{1}{1 + e^{-q}} \; where \; q = \frac{Inputs - \mu}{\sigma} \tag{5.11}$$

5.6.7. *Sigmoidal Normalization Hyperbolic Tangent (HT)*

The sigmoidal hyperbolic tangent (HT) normalization is based on the activation function tanh of the NN. HT is suitable for scaling the outliers by scaling the inputs linearly, without affecting the inputs.

$$Inputs' = \frac{1 - e^{-q}}{1 + e^{-q}} \; where \; q = \frac{Inputs - \mu}{\sigma} \tag{5.12}$$

5.7. Many-to-One RNN Architecture

An RNN with many-to-one architecture was trained for the linear dynamic model. The many-to-one RNN architecture is shown in figure 5.4 for a sequence length of three. The number of neurons in the hidden layer was equal to 10 with a tanh activation function, and an Adam optimizer was used for the RNN. The many-to-one RNN architecture uses the sequence length to estimate solutions, e.g., if the sequence length is three, then the many-to-one will analyze three values back in time to predict the fourth value. A one-to-many RNN architecture is useful if data features are dependent on each other and previous data are available.

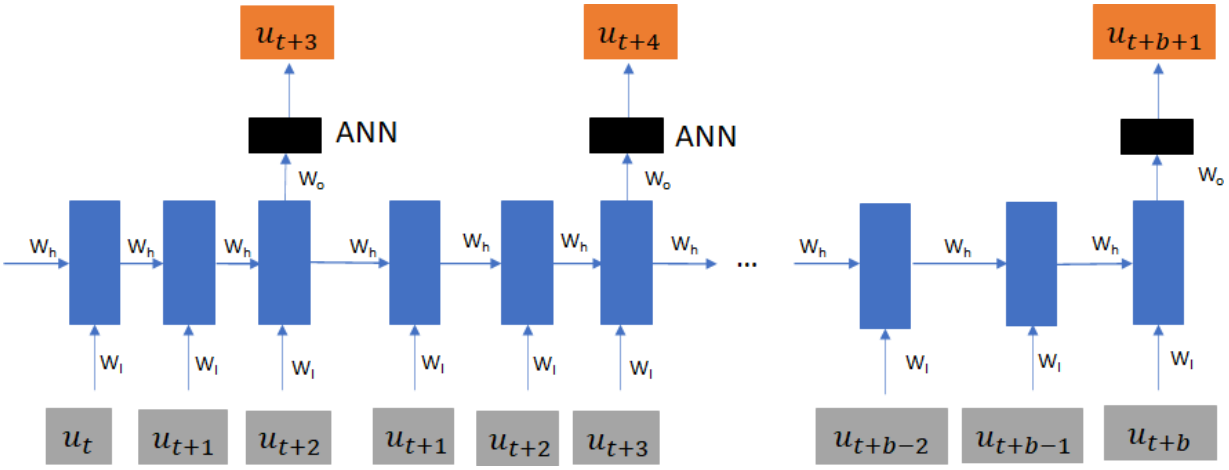

Figure 5.4: Many-to-one RNN architecture with sequence length = 3

Figure 5.5 shows the performance of the RNN on the training dataset with the different normalization techniques. The normalization technique that gave the smallest loss was VSS, followed by LS and TN. VSS had a very small training loss, as can be seen in figure 5.5, because the input values were small when normalized by VSS, and the loss between two very small values would be very small as well. However VSS did not perform well on the test set.

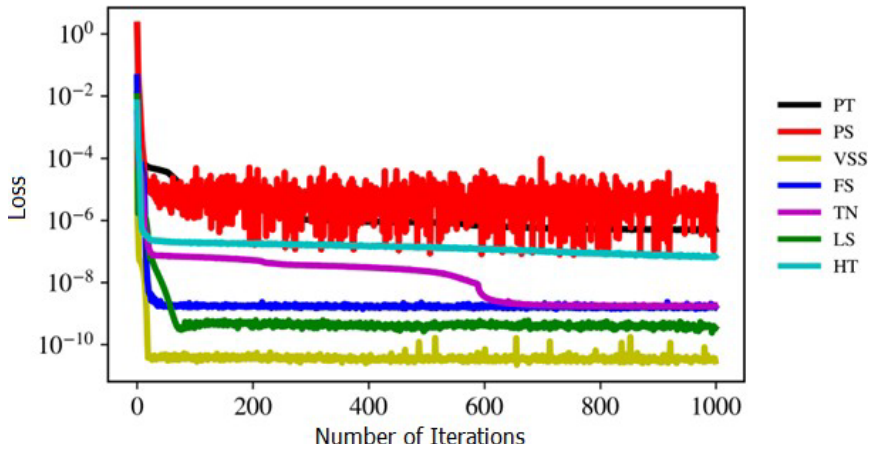

Figure 5.5: Training loss vs number of iterations for different normalization techniques

5.8. Results

After the RNN was trained, the different normalized trained RNNs were tested on a different dataset, called the test set. Figure 5.6 shows the results for the performance of the normalized RNNs on the test set.

The normalization technique that had the smallest test loss was TN. VSS, which performed well on the training set did not perform well on the test set for the reason mentioned above.

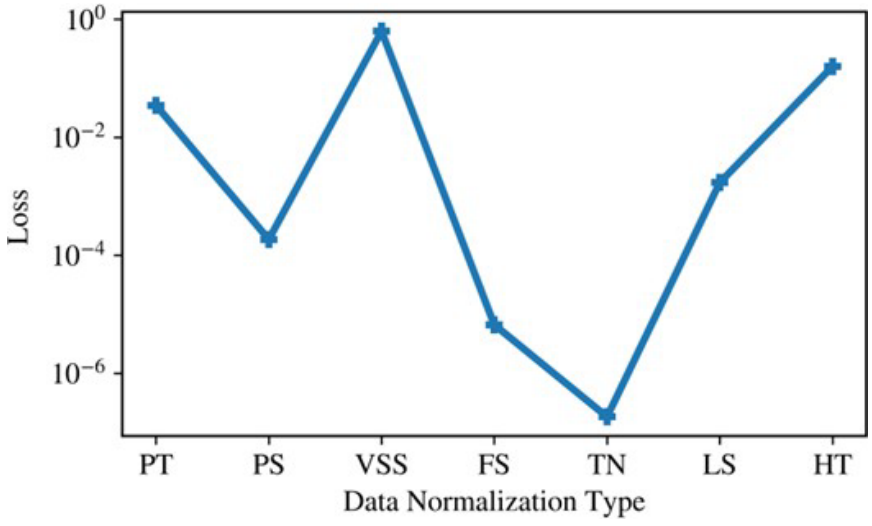

Figure 5.6: Test loss for different normalization techniques

An RNN was trained by using TN inputs with sequence length three, learning rate 1e-3, 10 neurons

in the hidden layers, and an Adam optimizer. Two test results of free vibration for the linear dynamics case, with mass $m = 1$ and stiffness $k = 1$, subjected to an initial displacement are shown in figure 5.7. The mean squared error loss recorded for 100 test files was 9.297e-07.

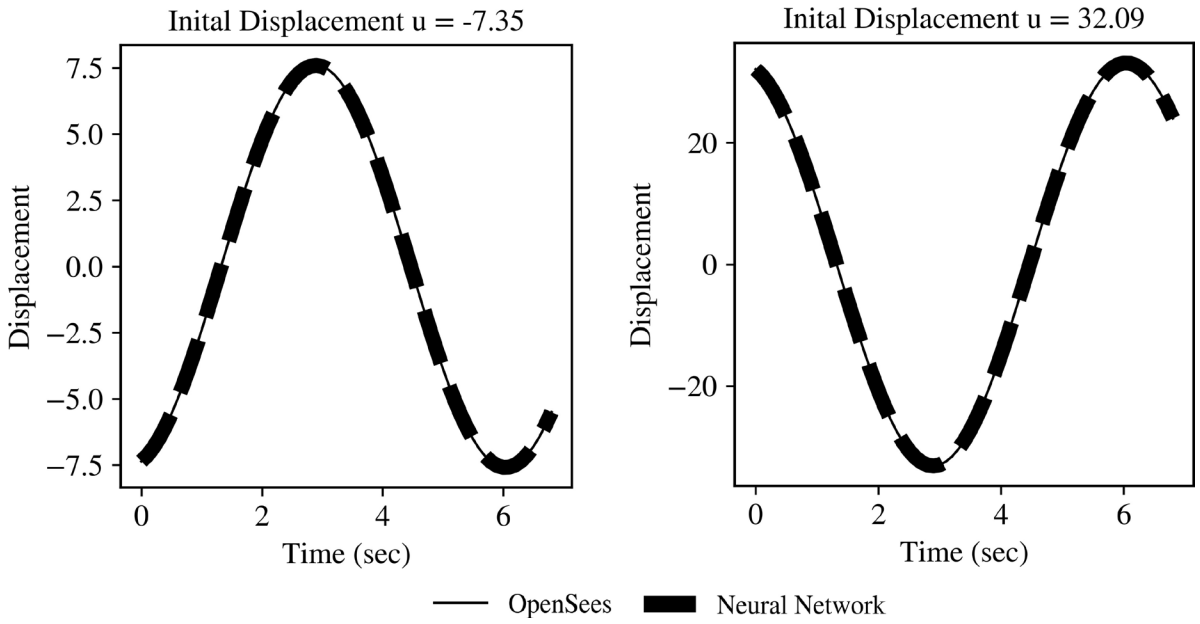

Figure 5.7: Free vibration results for two initial displacements values for the linear dynamic problem

5.9. Future Work

This chapter presented results for the data-driven RNN, which may not be robust or interpretable without the physical equations being embedded during learning. A physics-informed RNN should be studied under conditions in which the loss of the RNN includes the residual. variable mass, and stiffness values, nonlinear response, and multiple-degree-of-freedom systems should also be explored in future studies.

6. FLUID STRUCTURE INTERACTION

Some work has been done on using physics-informed neural networks for fluid structure interaction (FSI) problems, but such systems have been one dimensional and linear in nature. A study by (Raissi and Karniadakis 2017) used deep neural networks to deduce the velocity and pressure fields using the Navier-Stokes equations. The objective of the study was to transport an incompressible Newtonian passive scalar fluid in unbounded (external) and bounded (internal) geometries. Numerical time integration was performed on the Navier-Stokes equation and transport equations, and a small portion of data was selected for training the network. The success of the algorithm was based on recovering the flow velocity and pressure fields solely from the time series data. The results of the study were quite reasonable for a Newtonian, non-turbulent fluid and relatively noiseless and clean data. Another study by (White et al. 2019a) introduced a new method, the cluster network, with context networks and paired functions to overcome some limitations of the fully connected neural network to approximate full solutions. The cluster network has an inductive bias, which is stronger than the fully connected network to approximate the simple function and small number of local solutions.

This study preliminarily introduced and implemented NNs for an FSI problem using the particle finite element method (PFEM). The FSI problem was explored using two deep learning (DL) algorithms, including an artificial neural network (ANN) and a point cloud convolution network (PointConv). OpenSeesPy with the particle finite element method (PFEM) (M. Zhu and Scott 2014) was used to generate simulation data for training the neural networks.

6.1. PFEM and NN models

The structure shown in figure 6.1 is a highly flexible structure and includes fluid-structure interaction effects. In Figure 6.1, L is the length of the water, H is the height of the water, l is the distance of the water to the column, and b is the height of the column, which is the structure in this problem. These were used as the input parameters for the NN, and the output of the NN was the displacement of the column due to the fluid.

A PointConv (W. Wu et al. 2019) was also used to estimate FSI effects. A PointConv is a neural network that uses point convolution and point deconvolution layers and is used to predict 2D images. The term ”convolution” can be thought of as a visualization of an image by the neural network.

6.2. Results

Figure 6.2 shows the results of the displacement of the column from the NN and the PFEM models. The difference between the displacement peaks of the NN and the PFEM model was 0.082 percent, and

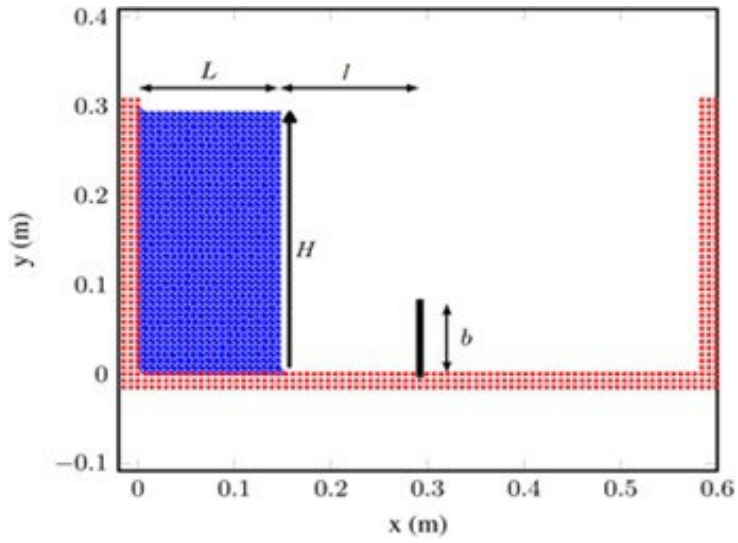

Figure 6.1: PFEM model with fluid structure interaction effects

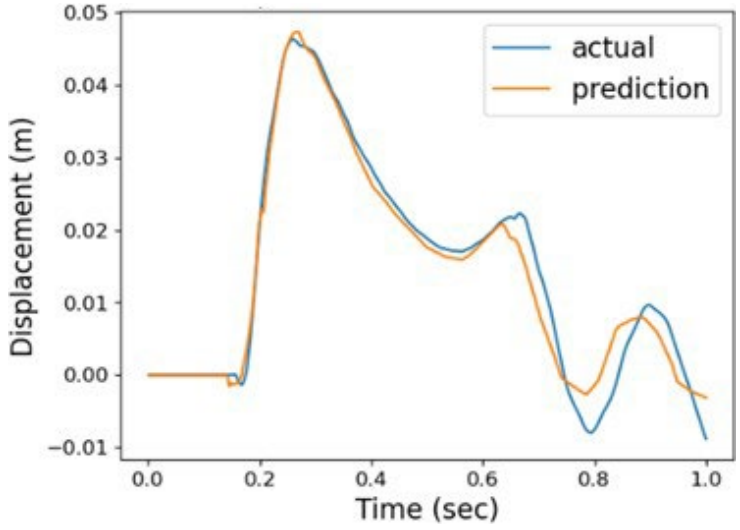

Figure 6.2: Column displacement results from the NN and PFEM models

the NN curve seemed to follow the PFEM model curve reasonably well.

As the fluid-structure interaction is a time series problem, PointConv was used to predict the pointwise state changes of the model in time tn, i.e., PointConv was trying to simulate the PFEM model. The blue points in figure 6.3 are PointConv-predicted simulations while the orange points are PFEM results from OpenSeesPy. PointConv clearly learned to avoid the fluid passing through boundaries and simulated the fluid in the correct direction, but the column displaced before coming into contact with the water and overall did not move as expected. The PointConv simulation needs more training time for a better

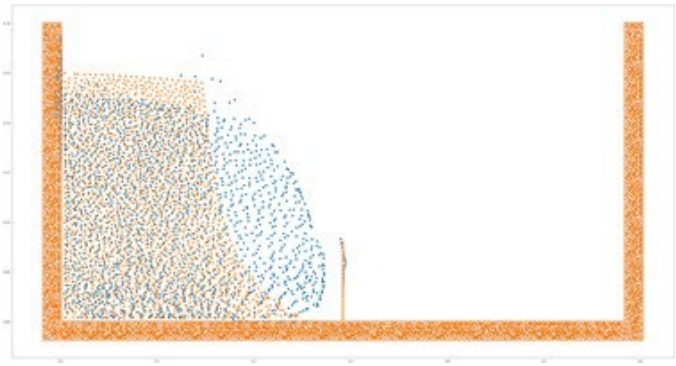

Figure 6.3: PointConv simulating the PFEM model

prediction of the PFEM model.

6.3. Summary and Conclusions

The performance of the algorithms presented in this chapter is promising. This chapter explores different types of ML/ DL algorithms without physics-informed approaches to estimate FSI effects. For the FSI problem, further investigation is still needed, including using the physics of PFEM during learning, evaluating different sets of inputs/ outputs, and investigating other types of NN architectures, such as time series prediction with the recurrent neural network instead of an ANN, as discussed in Chapter 5.

7. CONCLUSIONS

This pilot study applied machine learning (ML)/ deep learning (DL) algorithms to structural analysis problems, specifically for tsunami loading on bridges. Several data-driven and physics- informed approaches were compared to improve the reliability and interpretability of the NN.

To identify the best suited ML/ DL algorithms, the performances of several ML/DL algorithms were explored for simple structural engineering problems; i.e., linear-elastic static analysis and linear-elastic dynamic analysis of a single-degree-of-freedom (SDOF). For both linear-elastic static and dynamic problems, different NN hyperparameters were studied, tested, and selected based on best performance. An artificial neural network (ANN) was used for the static problem. A recurrent neural network (RNN) was used for the dynamic problem. The ANN and RNN had different optimal hyper parameters, such as batch size for the ANN and sequence length for RNN, which were tuned to find the best performance for the static and dynamic problems, respectively. A preliminary study using ANN and PointConv were used in conjunction with the particle finite element method (PFEM) in OpenSeesPy to estimate the effects of fluid-structure interaction (FSI).

7.1. Conclusions

The interpretability and reliability for the ML/ DL algorithms were evaluated by integrating the residual with the loss function. Several different types of loss functions were tested to assess performance in terms of the trained model's ability to estimate new test data and extrapolate beyond the training data domain. Conclusions are organized by chapter below:

- The physics-informed NN showed performance similar to that of the data-driven NN for the a static problem with constant stiffness. An NN with linear-like activation functions, i.e., linear and ReLU, had the best performance for both the physics-informed and data-driven cases, as they best matched the linear nature of the underlying physical equations, namely Hooke's Law. The trained ML/ DL model performed well on the extrapolation dataset, showing that the ML/DL model could generalize to new data conditions
- For a static model with varying stiffness, the NN algorithm produced good results when the data were log normalized to learn the linear nature of the physical equations. The physics-informed NN and data-driven NN showed similar performances, because the static model with varying stiffness was a linear problem, and the data-driven approach could provide a near-perfect fit to the data, provided that log normalization was utilized. The trained NN model performed well beyond the training dataset, and the NN can be used to estimate new data values.
- The dynamic model was investigated for data-driven RNN. A one-to-many RNN architecture was used to estimate the equations of motion. An RNN architecture was used so that the DL algorithm

could learn from previous values and find patterns to generalize to new values. The hyperbolic tangent normalization (TN) dataset was best suited for the RNN for the dynamics problem.

- Two DL models were evaluated to represent the nonlinearity of the FSI problem. An ANN was used to estimate displacement of the column that included FSI effects. Another DL model, known as PointConv, was used to simulate the particle finite element model (PFEM). Both results showed promise, but further investigation is needed, such as incorporating physics into the DL model and finding the best NN architecture for FSI problems.

7.2. Limitations and Future Work

The studies presented in this report were primarily for linear-elastic SDOFs and only preliminarily for FSI. This report contains only the data-driven part for the dynamic model and FSI problem. The point cloud convolution model, which represents the simulation for the FSI problem, needs hypertuning and more training time for better representation of the dataset.

Future work should include nonlinear problems, multiple-degree-of-freedom systems for the static and dynamic cases, and physics-informed NN models for the dynamic and FSI studies. Once refined, the ML/DL algorithm from the FSI studies could be helpful in overcoming the limitations of FSI, i.e., faster results and overcoming convergence issues to better understand tsunami loadings on bridges

REFERENCES

Adeli, Hojjat and C Yeh (1989). "Perceptron learning in engineering design". In: *Computer-Aided Civil and Infrastructure Engineering* 4.4, pp. 247–256.

Aladsani, Muneera, Henry Burton, Saman Abdullah, and John Wallace (May 2022). "Explainable Machine Learning Model for Predicting Drift Capacity of Reinforced Concrete Walls". In: *Aci Structural Journal* 119, pp. 191–204. DOI: 10.14359/51734484.

Alpaydin, Ethem (2020). *Introduction to machine learning*. MIT press.

Beucler, Tom, Michael Pritchard, Stephan Rasp, Jordan Ott, Pierre Baldi, and Pierre Gentine (2019). "Enforcing analytic constraints in neural-networks emulating physical systems". In: *arXiv preprint arXiv:1909.00912*.

Bibal, Adrien and Benoît Frénay (2016). "Interpretability of machine learning models and representations: an introduction." In: *ESANN*.

Bode, Mathis, Michael Gauding, Zeyu Lian, Dominik Denker, Marco Davidovic, Konstantin Kleinheinz, Jenia Jitsev, and Heinz Pitsch (2021). "Using physics-informed enhanced super-resolution generative adversarial networks for subfilter modeling in turbulent reactive flows". In: *Proceedings of the Combustion Institute* 38.2, pp. 2617–2625.

Bubba, Tatiana A, Gitta Kutyniok, Matti Lassas, Maximilian März, Wojciech Samek, Samuli Siltanen, and Vignesh Srinivasan (2019). "Learning the invisible: a hybrid deep learning-shearlet framework for limited angle computed tomography". In: *Inverse Problems* 35.6, p. 064002.

Carvalho, Diogo V, Eduardo M Pereira, and Jaime S Cardoso (2019). "Machine learning interpretability: A survey on methods and metrics". In: *Electronics* 8.8, p. 832.

Cha, Young-Jin, Wooram Choi, and Oral Büyüköztürk (2017). "Deep learning-based crack damage detection using convolutional neural networks". In: *Computer-Aided Civil and Infrastructure Engineering* 32.5, pp. 361–378.

Chen, Xinlei, Xiangxiang Xu, Xinyu Liu, Shijia Pan, Jiayou He, Hae Young Noh, Lin Zhang, and Pei Zhang (2018). "Pga: Physics guided and adaptive approach for mobile fine-grained air pollution estimation". In: *Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers*, pp. 1321–1330.

Coulibaly, Paulin and Connely K Baldwin (2005). "Nonstationary hydrological time series forecasting using nonlinear dynamic methods". In: *Journal of Hydrology* 307.1-4, pp. 164–174.

Daneshvar, Mohammad Hassan and Hassan Sarmadi (2022). "Unsupervised learning-based damage assessment of full-scale civil structures under long-term and short-term monitoring". In: *Engineering Structures* 256, p. 114059.

Doan, Nguyen Anh Khoa, Wolfgang Polifke, and Luca Magri (2019). “Physics-informed echo state networks for chaotic systems forecasting”. In: *International Conference on Computational Science*. Springer, pp. 192–198.

Dung, Cao Vu and Le Duc Anh (2019). “Autonomous concrete crack detection using deep fully convolutional neural network”. In: *Automation in Construction* 99, pp. 52–58.

Esmaeilzadeh, Soheil, Kamyar Azizzadenesheli, Karthik Kashinath, Mustafa Mustafa, Hamdi A Tchelepi, Philip Marcus, Mr Prabhat, Anima Anandkumar, et al. (2020). “Meshfreeflownet: A physics-constrained deep continuous space-time super-resolution framework”. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, pp. 1–15.

Flah, Majdi, Itzel Nunez, Wassim Ben Chaabene, and Moncef L Nehdi (2021). “Machine learning algorithms in civil structural health monitoring: a systematic review”. In: *Archives of Computational Methods in Engineering* 28.4, pp. 2621–2643.

Forssell, Urban and Peter Lindskog (1997). “Combining semi-physical and neural network modeling: An example ofits usefulness”. In: *IFAC Proceedings Volumes* 30.11, pp. 767–770.

Geneva, Nicholas and Nicholas Zabaras (2020). “Modeling the dynamics of PDE systems with physics-constrained deep auto-regressive networks”. In: *Journal of Computational Physics* 403, p. 109056.

Gilpin, Leilani H, David Bau, Ben Z Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal (2018). “Explaining explanations: An overview of interpretability of machine learning”. In: *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*. IEEE, pp. 80–89.

Glorot, Xavier and Yoshua Bengio (2010). “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, pp. 249–256.

Goodfellow, Ian, Yoshua Bengio, Aaron Courville, and Yoshua Bengio (2016). *Deep learning*. Vol. 1. 2. MIT press Cambridge.

Gulikers, Tom (2018). “An integrated machine learning and finite element analysis framework, applied to composite substructures including damage”. In:

Haghighat, Ehsan and Ruben Juanes (2020). “SciANN: A Keras/Tensorflow wrapper for scientific computations and physics-informed deep learning using artificial neural networks”. In: *arXiv preprint arXiv:2005.08803*.

Hampel, Frank R, Elvezio M Ronchetti, Peter J Rousseeuw, and Werner A Stahel (2011). *Robust statistics: the approach based on influence functions*. Vol. 196. John Wiley & Sons.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *CoRR* abs/1502.01852. arXiv: 1502.01852. URL: http://arxiv.org/abs/1502.01852.

Heusel, Martin, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Günter Klambauer, and Sepp Hochreiter (2017). “GANs Trained by a Two Time-Scale Update Rule Converge to a Nash Equilibrium”. In: *CoRR* abs/1706.08500. arXiv: 1706.08500. URL: http://arxiv.org/abs/1706.08500.

Jin, Kyong Hwan, Michael T McCann, Emmanuel Froustey, and Michael Unser (2017). “Deep convolutional neural network for inverse problems in imaging”. In: *IEEE Transactions on Image Processing* 26.9, pp. 4509–4522.

Kahana, Adar, Eli Turkel, Shai Dekel, and Dan Givoli (2020). “Obstacle segmentation based on the wave equation and deep learning”. In: *Journal of Computational Physics* 413, p. 109458.

Kani, J Nagoor and Ahmed H Elsheikh (2017). “DR-RNN: A deep residual recurrent neural network for model reduction”. In: *arXiv preprint arXiv:1709.00939*.

Karpatne, Anuj, William Watkins, Jordan Read, and Vipin Kumar (2017). “Physics-guided neural networks (pgnn): An application in lake temperature modeling”. In: *arXiv preprint arXiv:1710.11431*.

Karumuri, Sharmila, Rohit Tripathy, Ilias Bilionis, and Jitesh Panchal (2020). “Simulator-free solution of high-dimensional stochastic elliptic partial differential equations using deep neural networks”. In: *Journal of Computational Physics* 404, p. 109120.

Kharazmi, Ehsan, Zhongqiang Zhang, and George Em Karniadakis (2019). “Variational physics-informed neural networks for solving partial differential equations”. In: *arXiv preprint arXiv:1912.00873*.

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

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25, pp. 1097–1105.

LeCun, Yann, D Touresky, G Hinton, and T Sejnowski (1988). “A theoretical framework for back-propagation”. In: *Proceedings of the 1988 connectionist models summer school*. Vol. 1, pp. 21–28.

Loiseau, Jean-Christophe and Steven L Brunton (2018). “Constrained sparse Galerkin regression”. In: *Journal of Fluid Mechanics* 838, pp. 42–67.

M, Rekha (June 2020). *The Ascent of Gradient Descent*. URL: https://blog.clairvoyantsoft.com/the-ascent-of-gradient-descent-23356390836f.

Márquez-Neila, Pablo, Mathieu Salzmann, and Pascal Fua (2017). “Imposing Hard Constraints on Deep Networks: Promises and Limitations”. In: *CoRR* abs/1706.02025. arXiv: 1706.02025. URL: http://arxiv.org/abs/1706.02025.

McCulloch, Warren S and Walter Pitts (1943). “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.

Moraffah, Raha, Mansooreh Karami, Ruocheng Guo, Adrienne Raglin, and Huan Liu (2020). “Causal interpretability for machine learning-problems, methods and evaluation”. In: *ACM SIGKDD Explorations Newsletter* 22.1, pp. 18–33.

Moreira, Miguel and Emile Fiesler (1995). *Neural networks with adaptive learning rate and momentum terms*. Tech. rep. Idiap.

Nichols, James A, Hsien W Herbert Chan, and Matthew AB Baker (2019). "Machine learning: applications of artificial intelligence to imaging and diagnosis". In: *Biophysical reviews* 11.1, pp. 111–118.

Nielsen, Michael A (2015). *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA.

Paolucci, Roberto, Filippo Gatti, Maria Infantino, Chiara Smerzini, Ali Güney Özcebe, and Marco Stupazzini (2018). "Broadband ground motions from 3D physics-based numerical simulations using artificial neural networks". In: *Bulletin of the Seismological Society of America* 108.3A, pp. 1272–1286.

Papadrakakis, Manolis, Vissarion Papadopoulos, and Nikos D Lagaros (1996). "Structural reliability analyis of elastic-plastic structures using neural networks and Monte Carlo simulation". In: *Computer methods in applied mechanics and engineering* 136.1-2, pp. 145–163.

Papalambros, Panos Y and Douglass J Wilde (2000). *Principles of optimal design: modeling and computation*. Cambridge university press.

Parish, Eric J and Karthik Duraisamy (2016). "A paradigm for data-driven predictive modeling using field inversion and machine learning". In: *Journal of Computational Physics* 305, pp. 758–774.

Parker, DavidB (1985). "Learning-logic: Casting the cortex of the human brain in silicon". In:

Pathak, Deepak, Philipp Krähenbühl, and Trevor Darrell (2015). "Constrained Convolutional Neural Networks for Weakly Supervised Segmentation". In: *CoRR* abs/1506.03648. arXiv: 1506.03648. URL: http://arxiv.org/abs/1506.03648.

Peng, Hong, Jingwen Yan, Ying Yu, and Yaozhi Luo (2021). "Time series estimation based on deep Learning for structural dynamic nonlinear prediction". In: *Structures*. Vol. 29. Elsevier, pp. 1016–1031.

Raissi, Maziar and George E. Karniadakis (2017). "Hidden Physics Models: Machine Learning of Nonlinear Partial Differential Equations". In: *CoRR* abs/1708.00588. arXiv: 1708.00588. URL: http://arxiv.org/abs/1708.00588.

Raissi, Maziar, Paris Perdikaris, and George E Karniadakis (2019). "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378, pp. 686–707.

Raissi, Maziar, Zhicheng Wang, Michael S Triantafyllou, and George Em Karniadakis (2019). "Deep learning of vortex-induced vibrations". In: *Journal of Fluid Mechanics* 861, pp. 119–137.

Raissi, Maziar, Alireza Yazdani, and George Em Karniadakis (2018). "Hidden fluid mechanics: A navier-stokes informed deep learning framework for assimilating flow visualization data". In: *arXiv preprint arXiv:1808.04327*.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, p. 386.

San, Omer and Romit Maulik (2018). “Machine learning closures for model order reduction of thermal fluids”. In: *Applied Mathematical Modelling* 60, pp. 681–710.

Sanchez-Gonzalez, Alvaro, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia (2020). “Learning to simulate complex physics with graph networks”. In: *International Conference on Machine Learning*. PMLR, pp. 8459–8468.

Senouf, Ortal, Sanketh Vedula, Tomer Weiss, Alex Bronstein, Oleg Michailovich, and Michael Zibulevsky (2019). “Self-supervised learning of inverse problem solvers in medical imaging”. In: *Domain adaptation and representation transfer and medical image learning with less labels and imperfect data*. Springer, pp. 111–119.

Shah, Viraj, Ameya Joshi, Sambuddha Ghosal, Balaji Pokuri, Soumik Sarkar, Baskar Ganapathysubramanian, and Chinmay Hegde (2019). “Encoding invariances in deep generative models”. In: *arXiv preprint arXiv:1906.01626*.

Singh, Dalwinder and Birmohan Singh (2020). “Investigating the impact of data normalization on classification performance”. In: *Applied Soft Computing* 97, p. 105524.

Sirignano, Justin and Konstantinos Spiliopoulos (2018). “DGM: A deep learning algorithm for solving partial differential equations”. In: *Journal of computational physics* 375, pp. 1339–1364.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1, pp. 1929–1958.

Sun, Luning, Han Gao, Shaowu Pan, and Jian-Xun Wang (2020). “Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data”. In: *Computer Methods in Applied Mechanics and Engineering* 361, p. 112732.

Thai, Huu-Tai (2022). “Machine learning for structural engineering: A state-of-the-art review”. In: *Structures*. Vol. 38. Elsevier, pp. 448–491.

Thompson, Michael L and Mark A Kramer (1994). “Modeling chemical processes using prior knowledge and neural networks”. In: *AIChE Journal* 40.8, pp. 1328–1340.

Ulyanov, Dmitry, Andrea Vedaldi, and Victor Lempitsky (2018). “Deep image prior”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 9446–9454.

Wang, Sifan, Yujun Teng, and Paris Perdikaris (2021). “Understanding and mitigating gradient flow pathologies in physics-informed neural networks”. In: *SIAM Journal on Scientific Computing* 43.5, A3055–A3081.

White, Cristina, Daniela Ushizima, and Charbel Farhat (2019a). *Fast Neural Network Predictions from Constrained Aerodynamics Datasets*. DOI: 10.48550/ARXIV.1902.00091. URL: https://arxiv.org/abs/1902.00091.

White, Cristina, Daniela Ushizima, and Charbel Farhat (2019b). “Neural networks predict fluid dynamics solutions from tiny datasets”. In: *arXiv preprint arXiv:1902.00091*.

Willard, Jared, Xiaowei Jia, Shaoming Xu, Michael Steinbach, and Vipin Kumar (2020). “Integrating scientific knowledge with machine learning for engineering and environmental systems”. In: *arXiv preprint arXiv:2003.04919*.

Wu, Jin-Long, Karthik Kashinath, Adrian Albert, Dragos Chirila, Heng Xiao, et al. (2020). “Enforcing statistical constraints in generative adversarial networks for modeling chaotic dynamical systems”. In: *Journal of Computational Physics* 406, p. 109209.

Wu, Wenxuan, Zhongang Qi, and Li Fuxin (2019). “Pointconv: Deep convolutional networks on 3d point clouds”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9621–9630.

Yang, Liu, Sean Treichler, Thorsten Kurth, Keno Fischer, David Barajas-Solano, Josh Romero, Valentin Churavy, Alexandre Tartakovsky, Michael Houston, Mr Prabhat, et al. (2019). “Highly-scalable, physics-informed GANs for learning solutions of stochastic PDEs”. In: *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, pp. 1–11.

Yang, Yibo and Paris Perdikaris (2019). “Adversarial uncertainty quantification in physics-informed neural networks”. In: *Journal of Computational Physics* 394, pp. 136–152.

Yang, Yibo and Paris Perdikaris (2018). “Physics-informed deep generative models”. In: *arXiv preprint arXiv:1812.03511*.

Yin, Wenpeng, Katharina Kann, Mo Yu, and Hinrich Schütze (2017). “Comparative study of CNN and RNN for natural language processing”. In: *arXiv preprint arXiv:1702.01923*.

Zadeh, Reza (Nov. 2016). *The hard thing about deep learning*. URL: https://www.oreilly.com/radar/the-hard-thing-about-deep-learning/?twitter=%40bigdata.

Zhang, Liang, Gang Wang, and Georgios B Giannakis (2019). “Real-time power system state estimation and forecasting via deep unrolled neural networks”. In: *IEEE Transactions on Signal Processing* 67.15, pp. 4069–4077.

Zhang, Linfeng, Jiequn Han, Han Wang, Roberto Car, and E Weinan (2018). “Deep potential molecular dynamics: a scalable model with the accuracy of quantum mechanics”. In: *Physical review letters* 120.14, p. 143001.

Zhu, Minjie and Michael H Scott (2014). “Improved fractional step method for simulating fluid-structure interaction using the PFEM”. In: *International Journal for Numerical Methods in Engineering* 99.12, pp. 925–944.

Zhu, Yinhao, Nicholas Zabaras, Phaedon-Stelios Koutsourelakis, and Paris Perdikaris (2019). “Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data”. In: *Journal of Computational Physics* 394, pp. 56–81.