Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Torkel Laache

Physics Guided Machine Learning: Injecting neural networks with simplified theories

Master's thesis in Cybernetics and Robotics
Supervisor: Adil Rasheed
May 2021

NTNU
Kunnskap for en bedre verden

Torkel Laache

Physics Guided Machine Learning: Injecting neural networks with simplified theories

Master's thesis in Cybernetics and Robotics
Supervisor: Adil Rasheed
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Abstract

Exponential growth in computing power and availability of large datasets has popularized and progressed machine learning substantially in recent years. Neural networks are potent approximators capable of discovering patterns in complex datasets and possibly model realistic dynamical systems. This thesis investigates a physics-guided machine learning framework of neural networks that combines traditional mathematical modeling with machine learning methods. Here, neural networks are injected with simplified theories of dynamical systems at intermediate layers to improve their accuracy and interpretability. To validate the framework, it undergoes several experiments on various systems such as the Lotka-Volterra equations, Duffin oscillator, Lorenz system, Hindmarsh–Rose model, and Willamowski-Rössler model. The results indicate that the proposed framework can enhance the neural networks and be used in various scientific machine learning applications, particularly in systems where simplified theories can guide the learning process.

Sammendrag

Eksponentiell vekst i datakraft og tilgjengelighet av store datasett har popularisert og forbedret maskinlæring betydelig de siste årene. Nevrale nettverk er sterke verktøy som kan oppdage mønstre i komplekse datasett og muligens modellere realistiske dynamiske systemer. Denne oppgaven undersøker et "physics-guided machine learning" rammeverk for nevrale nettverk ved å kombinere tradisjonell matematisk modellering med maskinlæringsmetoder. Her injiseres nevrale nettverk med forenklede teorier om dynamiske systemer for å forbedre nøyaktigheten og tolkningen. For å validere rammeverket gjennomgår det flere eksperimenter på forskjellige systemer som Lotka-Volterra-ligningene, Duffing-oscillatoren, Lorenz-systemet, Hindmarsh-Rose modellen og Willamowski-Rössler modellen. Resultatene indikerer at det foreslåtte rammeverket kan forbedre nevrale nettverk og brukes i forskjellige vitenskapelige maskinlæringsapplikasjoner, spesielt i systemer der en forenklet modell kan hjelpe læringsprosessen.

Preface

This thesis concludes my Master's degree in Cybernetics and Robotics at the Norwegian University of Science and Technology, under the supervision of Adil Rasheed during the spring of 2021. Even though it is not a direct extension of the specialization project completed in the autumn of 2020, the experience and background knowledge with machine learning has been utilized.

In addition to writing a master thesis with no direct link to the specialization project, the thesis topic changed during the semester. Nonetheless, the previous topic was also regarding machine learning, so it was not too challenging. Before the specialization project and thesis, I had little experience with machine learning. Fortunately, many helpful tools exist, such as the python library Keras, which most of the experiments utilized. Many different experiments were evaluated, but due to a lack of proper equipment, some of the earlier ones would take upwards of 20 hours and ending up misusing time.

I would like to express my gratitude to Professor Adil Rasheed, my supervisor, for his guidance and support throughout the thesis. I am also grateful to Erlend Lundby for the helpful discussions on this subject. Additionally, it would be much harder to finish this thesis without my roommate Ruben Brecke, who kept my spirits up in these troublesome times

31.05.2021

Torkel Laache

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

AI	Artificial Intelligence
AN	Artificial Neuron
ANN	Artificial Neural Network
CFD	Computational Fluid Dynamics
DNN	Deep Neural Network
FNN	Feedforward Neural Network
ML	Machine Learning
NN	Neural Network
ODE	Ordinary Differential Equation
PGML	Physics Guided Machine Learning
PGNN	Physics Guided Neural Network
RK45	Runge-Kutta method of 4(5) order
RNN	Recurrent Neural Network
ROM	Reduced Order Models

Symbols

α_i	Lotka-Volterra parameters
β_i	Duffing parameters

δ	Neuron error
η	Learning rate
γ_i	Chaotic systems parameters
ω	Neuron weight
ρ	Activation function threshold
σ	Activation function
θ	Network parameters
b	Neuron bias
L	Loss function
t	Time
x_i	Inputs
y_i	Outputs

1 Introduction

With the availability of large datasets, coupled with exponential growth in computing power and improvements in algorithms, the interest in machine learning has had unprecedented growth in recent years. Powerful tools such as artificial neural networks (ANN) can approximate complex functions and systems, possibly replacing traditional methods in the future. However, interpreting the black-box structure of ANNs is non-trivial, restricting its use for critical real-world applications.

1.1 Motivation and Background

Up until recent years, modeling dynamical systems with mathematical functions has been the norm. To some extent, mathematical functions can represent everything from chemistry to electrical engineering to economics. However, it requires a trade-off between the model's accuracy and simplicity. Realistic models are complex and demanding to analyze and comprehend, and can also pose computational issues like long run time and numerical instability. Machine learning (ML) has proven to be a handy tool in many engineering fields in recent years. With the current rate of progress in ML, ML-based modeling might replace the traditional methods.

ANNs are a subset of ML that has made tremendous progress lately. They are a class of *universal approximators* [1] capable of approximating any function and dynamical system. However, as the ANNs increase complexity, the amount of parameters rapidly grows, compromising their interpretability and reliability. While they allow for robust predictions, their black-box nature lack transparency and cannot be fully explained [2].

In this thesis, we investigate a physics-guided machine learning (PGML) framework where ANNs are injected with simplified theories of a system at intermediate layers. Injecting the simplified theories would allow the network to relate to or bridge the gap between the simplified theories and the complete systems. It will also help gain insight into how the network trains and its structure, increasing their interpretability. Unlike some other methods that only aid the training phase, this framework will also support the network during prediction.

1.1.1 State of the art

Modeling dynamical systems with ANNs has had solid results for many different tasks, and various techniques incorporating prior scientific knowledge about the system into the networks have also succeeded. In [3], the physical relationships between the temperature, density, and depth of water are used to design a custom loss function for the network to minimize when training. Although the technique improves models in many cases [4, 5], designing custom loss functions can be demanding, and it only assists the network during the training phase.

Another technique is reduced-order models (ROM). These models project a complete system to a system of fewer degrees of freedom (DOF) that encapsulates most, if not all, of the system's fundamental dynamics [6]. ROMs achieve significant improvements in computational efficiency. However, it requires a complete description of the system's dynamics, which is frequently unknown or insufficient for the desired purpose. Furthermore, they often lack robustness regarding parameter changes and must usually be rebuilt for each parameter variation [7]. Nevertheless, ROMs have proved successful in many cases [8, 9, 10, 11].

While there are others methods [12, 13, 14] than those mentioned, this thesis builds upon the framework introduced in [15], tested on a canonical airfoil aerodynamic problem. Typically, the flow around an airfoil can be predicted using computational fluid dynamics (CFD). This problem is a nonlinear, high-dimensional, and multiscale problem that becomes computationally intractable when the design space increases. To solve this, they combined CFD and ML

by injecting parameters from simplified CFD theories at intermediate layers in a NN. The method significantly reduced uncertainty in performance, showing great potential for scientific machine learning.

1.2 Objectives

Primary Objective: Investigate the PGML framework and evaluate its performance metrics on parts such as training, prediction, and generalization.

Doing so, the project aims to answer the following research questions:

- Does the proposed framework improve performance, in both accuracy and interpretability?
- What types of prior knowledge must be injected before improvements occur? Are there specific types of functions, and how much information is needed?

1.3 Outline of Report

The report starts with the motivation to take on this thesis and introduces some of the technology used. Chapter 2 addresses the background theory on machine learning, emphasizing neural networks. In Chapter 3, an overview of the various experiments, including how the hyperparameters were chosen, specifications, and other options will be presented. Chapter 4 contains the results and discussion from the experiments. Finally, Chapter 5 wraps up the thesis and offers some suggestions for future research.

2 Theory

This chapter focuses on the background theory required for this project and justifies the methods used in Chapter 3. Machine learning is the first topic discussed, and because it is such a broad topic, only the most critical aspects are introduced. After, the structure of artificial neural networks and their parameters are discussed, validating some of the later choices.

2.1 Machine Learning

Machine learning is a subset of Artificial Intelligence where computers automatically learn and improve through experience and can make decisions and predictions without being explicitly programmed. While it has existed since the 1940s, it only gained popularity in recent years when computers' speed and computational power improved and larger datasets were made available.

ML algorithms can be broadly categorized as supervised and unsupervised learning, by what kind of experience they are allowed to have during the learning process [16]. In supervised learning, algorithms attempt to learn a function that maps features x to target y based on example input-output pairs $\{(x_1, y_1), ..., (x_n, y_n)\}$. In contrast, unsupervised learning algorithms attempt to draw inferences with unknown targets, making them less intuitive. This categorization is not decisive, as there are other categories such as semi-supervised learning and reinforcement learning. However, supervised methods are the most widely used [17] and the one used in this thesis.

Given the example input-output pairs (x_i, y_i), supervised learning attempts to approximate the function $f(x_i)$ that can map inputs x_i to outputs y_i. Furthermore, the function should generalize to unseen data. So one often withholds

a *test set* during the training phase to evaluate the algorithm's ability to generalize. If not, problems like *overfitting* occur whenever the algorithm performs notably worse on the test set compared to the training set, simply remembering examples instead of noticing patterns. Numerous techniques exist to avoid overfitting, such as early stopping or dropouts for neural networks[18, 19], which should be chosen depending on the task.

Most machine learning algorithms are parameterized by a set of *hyperparameters*, allowing tuning of the algorithm based on prior knowledge. Hyperparameter tuning is a challenging task, often performed manually by trial and error, testing different sets of hyperparameters on a predefined grid [20], or via rules of thumb [21]. An algorithm's success largely depends on the hyperparameters, where small changes can lead to significant variance. Ideally, one should minimize the number of tunable hyperparameters to avoid poor reproducibility [22] or suffer from the *curse of dimensionality* [23]. It is crucial to avoid *peeking* at the test set during the tuning process, as it would invalidate the results. Improving the algorithm's performance on the test set would leak information into the algorithm, corrupting the experiment. Therefore, one often withholds an additional part of the training set for validation, often called a *validation set*.

Supervised learning tasks are separated into two types, *classification* and *regression* tasks. In classification tasks, the algorithm maps the input data to a set of discrete values (e.g., "True" or "False"), labeling the data based on recognized patterns. Regression tasks involve approximating a continuous output value (e.g., weight 0-100g), typically used in forecasting, predicting, and finding a relationship between data. For both tasks, the algorithm predicts an output $y_i = f(x_i)$ and updates its parameters based on feedback through a loss function $L(y_i, y_i)$.

2.2 Artificial neural network

With their proofs of convergence and relatively simple designs, traditional ML methods have many applications. However, when either the dimensionality or complexity of the data becomes too large, their utility is limited. ANNs are modeled to approximate any function to improve upon these methods, overlooking

the dimensionality and data complexity. Compared to the traditional methods, ANNs perform better at forecasting due to their ability to capture hidden, non-linear trends the traditional methods can not [24, 25]. As approximating complex functions need large quantities of data and computational power, ANNs were only favored in recent years when such resources were made more available.

2.2.1 Artificial neurons

An ANN can be considered a computing system vaguely inspired by the structure of biological neural networks, such as the human brain. The system comprises interconnected processing units called artificial neurons (AN), whose general model consists of a summing part and an output part [26]. An artificial neuron has one or more inputs $(x_1, x_2, ..., x_n)$ that are separately weighted with weights $(w_1, w_2, ..., w_n)$, producing a weighted sum as illustrated in Figure 2.2.1. As the network is interconnected, the inputs may come from other ANs or external sources. The weighted sum is passed through an activation function, which decides whether an artificial neuron is activated or not. If $u = \sum_{i=1}^{n} x_i w_i$ and ρ is the threshold for the activation function, the output y is:

$$y = \begin{cases} 1 & \text{if } u \geq \rho \\ 0 & \text{if } u < \rho \end{cases} \tag{2.2.1}$$

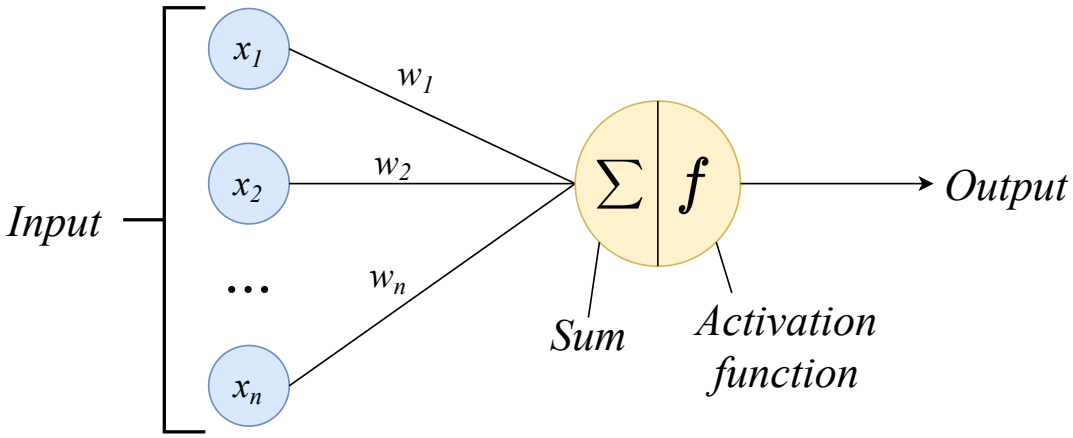
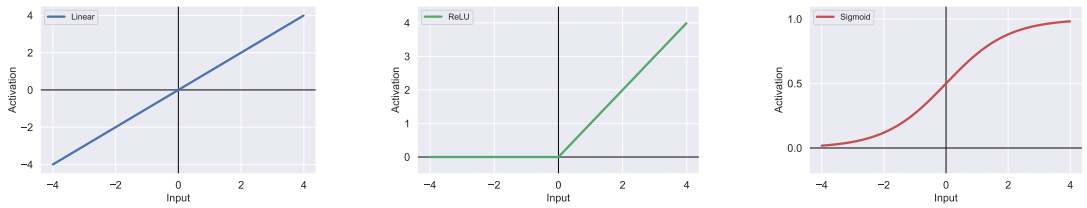


Figure 2.2.1: An artificial neuron

Activation functions are vital as they support understanding and learning complex mappings between corresponding inputs and outputs. Linear activation

functions, like Equation 2.2.1, can only adapt to linear changes. Nonlinear activation functions are preferred over linear activation functions because errors in the real world have nonlinear characteristics [27]. *A sufficiently large ANN using nonlinear activation functions can approximate arbitrarily complex functions* [28] .

Nonlinear activation functions tend to be computationally expensive, so it is common to select a simple nonlinear function. There are numerous good activation functions, such as the Sigmoid and Tanh functions, so the choice typically depends on the task. However, the Rectified Linear Unit (ReLU) has recently become very popular as it provides fast and effective training on complex data while reducing the chance of suffering from the *vanishing gradient problem*. It is relatively simple since it behaves purely linearly when the input is greater than zero, as shown in Figure 2b. Often, a combination of activation functions is used for a single network. For instance, a binary classification task requires an output of 0 or 1. In that case, instead of only using ReLU, the Sigmoid would replace it at the output. For regression tasks, the output can range between -inf and inf; thus, the linear activation function in Figure 2a would be used for output.



(2a) Linear **(2b)** ReLU **(2c)** Sigmoid

Figure 2.2.2: Activation functions

2.2.2 Neural network

As illustrated in Figure 2.2.3, ANN are organized in multiple layers, with each layer consisting of several interconnected neurons. The artificial neurons in the *input layer* are activated through the initial data, ranging from raw pixels in an image to sensor readings. The *hidden layer* neurons are activated via weighted

connections from the input layer and previously active neurons until they reach the *output layer*. Adding more layers and neurons can help the ANN express increasingly complex functions, and when there is more than one hidden layer, it is often referred to as a deep neural network (DNN).

The hidden layers in a DNN are often referred to as a *black box*, as studying the DNN's structure provides no insight into the approximated function structure. Understanding the relationship between the weights and the approximated function remains a mystery, raising safety concerns about whether DNN is ready to make automated decisions on critical human-related matters. However, some tools can assist in providing insight [29, 30].

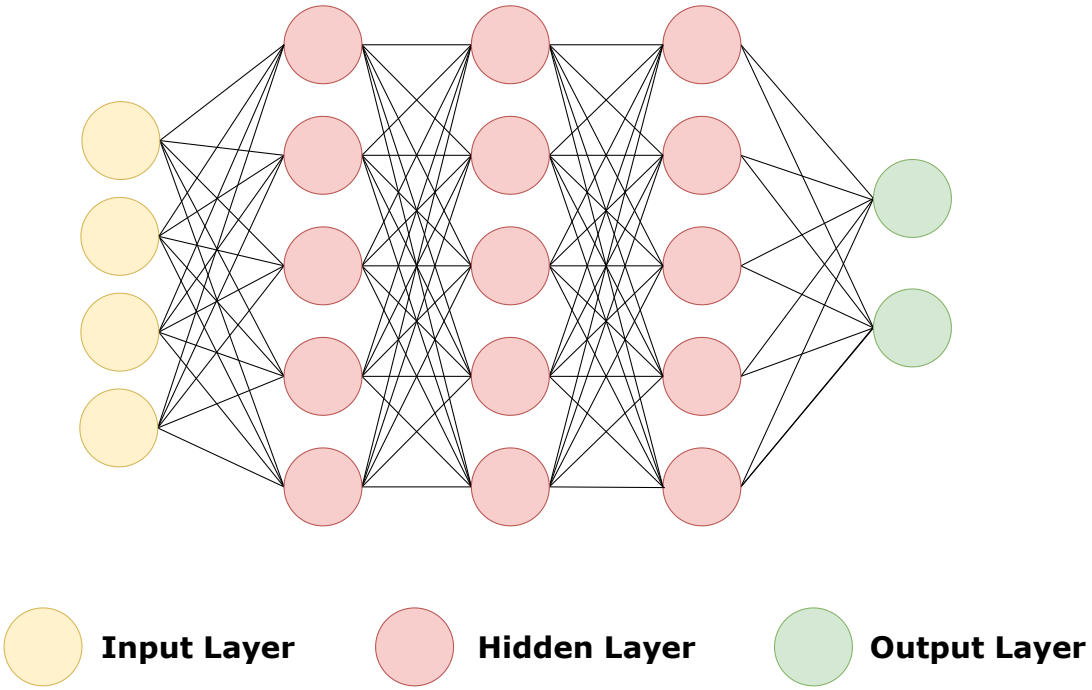


Figure 2.2.3: An artificial neural network with three hidden layers.

There are several steps that an ANN must take in order to learn. In supervised learning, the data is passed through the network in a process known as *forward propagation*, where the network attempts to process the data from input to output. To evaluate how well the ANN has approximated the actual function, a measure of error is necessary. Loss functions provide feedback to the network by measuring the error between the network's output y and the true output

$\hat{y}$. These are typically represented as $L(y, \hat{y})$, where L: $\mathbb{R}^n \rightarrow \mathbb{R}$ denotes some measure of error. For regression tasks, the mean squared error (MSE), as shown in Equation 2.2.2, is often chosen as the loss function.

$$L = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2 \tag{2.2.2}$$

Approximating the true function is equivalent to minimizing the loss function. Minimizing the loss can be done by gradient descent, using the gradient of the loss $\nabla L(\theta)$ with respect to the trainable parameters θ. The parameters (weights and biases) are updated iteratively at a rate called *learning rate* η, as shown in Function 2.2.3. Choosing a proper learning rate can be difficult, as too small leads to painfully slow convergence, while too large can fluctuate around the minimum or even diverge [31]. One solution is to compute adaptive learning rates for each parameter where the learning rate decays over time, taking large steps at the beginning and smaller steps towards the end.

$$\theta_{i+1} \leftarrow \theta_i - \eta \nabla L(\theta_i) \tag{2.2.3}$$

2.2.3 Backpropagation

The backpropagation algorithm is the cornerstone of learning in ANNs. It efficiently computes the gradient of the loss function with respect to the local parameters (weights and biases), in contrast to a direct computation of the gradient for each parameter individually. The gradient will inform how quickly the loss changes when the parameters are adjusted and how their changes affect the network's overall behavior. Most of the theory and notations for the backpropagation algorithm are from [32].

Some notation is required to describe individual weights and biases in an arbitrary ANN before deriving the relevant equations for the backpropagation algorithm. This can be found in Table 2.2.1. Instead of using the loss L, it will be replaced with the cost C, as L denotes the output layer here. Also, whenever the j subscript is removed from any of the terms, for example b^l_j, it is in its matrix form. For example, b^l represents the vector containing biases for the neurons in layer l.

Term	Other form	Definition
w^l_{jk}		The weight for the connection from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer
z^l_j	$(\sum_k w^l_{jk} a^{l-1}_j) + b^l_j$	Weighted input to the j^{th} neuron in the l^{th} layer
σ		Activation function
a^l_j	$\sigma(z^l_j)$	Activation of the j^{th} neuron in the l^{th} layer
δ^l_j	$\frac{\partial C}{\partial z^l_j}$	Error in the j^{th} neuron in the l^{th} layer
b^l_j		Bias of the j^{th} neuron in the l^{th} layer

Table 2.2.1: Notation for backpropagation algorithm based on theory from [32].

Using the notation in Table 2.2.1, we can derive the equations for the algorithm. For a single neuron j in the output layer L, its error can be calculated using the chain rule as

$$\delta^L_j = \frac{\partial C}{\partial z^L_j} = \frac{\partial C}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j} = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j) \tag{2.2.4}$$

The backpropagation algorithm requires it in its matrix-based form, which is

$$\delta^L = \nabla_{a^L} C \odot \sigma'(z^L) \tag{2.2.5}$$

where $\odot$ represents the *elementwise* product, often called the Hadamard product. Finding the error in the output layer makes it possible to find the error in the previous layer, as shown below.

$$\begin{aligned}\delta_j^{L-1} &= \frac{\partial C}{\partial z_j^{L-1}} \\ &= \sum_k \frac{\partial C}{\partial z_k^L}\frac{\partial z_k^L}{\partial z_j^{L-1}} \\ &= \sum_k \delta_k^L \frac{\partial z_k^L}{\partial z_j^{L-1}} \\ &= \sum_k w_{kj}^L \delta_k^L \sigma'(z_j^{L-1})\end{aligned}$$

This will also be represented in it matrix-based form as

$$\delta^{L-1} = ((w^L)^T \delta^L) \odot \sigma'(z^{L-1}) \tag{2.2.6}$$

In equations 2.2.5 and 2.2.6 is where the vanishing gradient problem occurs. As the error is dependent on the first derivative of the activation function, certain activation functions can impose a problem. Take the Sigmoid Function 2c from earlier. If the input is large enough, the output has a nearly flat slope, and the derivative is close to zero. Multiplying these small derivatives as it backpropagates the network results in a gradient that decreases exponentially. A neuron that exhibits these characteristics is said to be saturated, and as a result, learns very slowly. For the remaining layer, the errors are found by backpropagating δ^L like:

$$\begin{aligned}\delta^{L-1} &= ((w^L)^T \delta^L) \odot \sigma'(z^{L-1}) \\ \delta^{L-2} &= ((w^{L-1})^T \delta^{L-1}) \odot \sigma'(z^{L-2}) \\ &\vdots \\ \delta^{L} &= ((w^{L+1})^T \delta^{L+1}) \odot \sigma'(z^{L})\end{aligned}$$

The loss across the weights and biases in the network can then be distributed using these errors:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1} \tag{2.2.7}$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} = \delta_j^l \tag{2.2.8}$$

Finally, the backpropagation algorithm can be defined as

Algorithm 1: Backpropagation algorithm

1. **Initialize** ANN with random weights and biases
2. **Input** x: Set the corresponding activation a^1 for the input layer.
3. **Feedforward:** For each $l \in \{2, 3, ..., L\}$, compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$
4. **Output error:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$
5. **Backpropagate the error:** For each layer $l \in \{L-1, L-2, ..., 2\}$, compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
6. **Output:** The gradient of the cost function is given by $\frac{\delta C}{\delta w^l_{jk}} = a^{l-1}_k$ and $\frac{\delta C}{\delta b^l_j} = \delta^l_j$

Algorithm 1 produces an output well suited for iterative optimization algorithms such as gradient descent. Combined, they allow an ANN to iteratively reduce its error and approximate a function that maps the input-output pairs. Gradient descent will rarely find the global minimum, as nearly any ANN is virtually guaranteed to have a vast number of local minima. *Experts now suspect that, for sufficiently large neural networks, most local minima have a low loss function value, and that it is not important to find the true global minimum* [16].

3 Method and set-up

This chapter describes the method and experiments done to achieve the results presented in Chapter 4. First, the physics-guided neural networks' (PGNN) framework will be introduced, as well as the data generation process. Several systems will be tested to investigate the robustness and capabilities of the framework in various situations. Lastly, some specifications and parameter choices will be justified. The code can be found at `https://github.com/sjokkopudd/PG-NN`

3.1 Physics-Guided Neural Network

Comprehensive models of dynamical systems are complex and often take extensive computational run time, limiting their use in many cases where a model run is required at each iteration. Using neural networks (NN) instead could reduce computational run time as long as it performs at a high enough standard. Increasing the accuracy and reducing the training time for NN is of great importance, especially when they are to model very complex systems. The PGNN framework presented in this paper hopes to improve the accuracy, training time, and interpretability using a simple architecture.

The basic idea of the PGNN is to use known knowledge from a dynamical system to assist the NN by injecting it at an intermediate layer. The known knowledge could be information from a simplified physics-based model of the whole system, such as [15], or part of the actual system. Given a dynamical system controlled by the ODE on the form

$$\dot{x(t)} = f(t, x)$$

Assuming $f(t, x)$ is a very complex system, but a simplified model $h(x)$ based on known knowledge exists so

$$\dot{x(t)} \simeq h(t, x)$$

Then it is reasonable to assume that the complex system $f(x)$ can be described as

$$f(t, x) = F(t, x, h(t, x))$$

where F is a function of lower complexity than f. The idea is that feeding $h(x)$ to the NN could assist the NN and bring interpretability to otherwise black-box models. The proposed method, illustrated in Figure 3.1.1, is adaptable to a wide range of physical systems and could have significant potential in scientific machine learning.

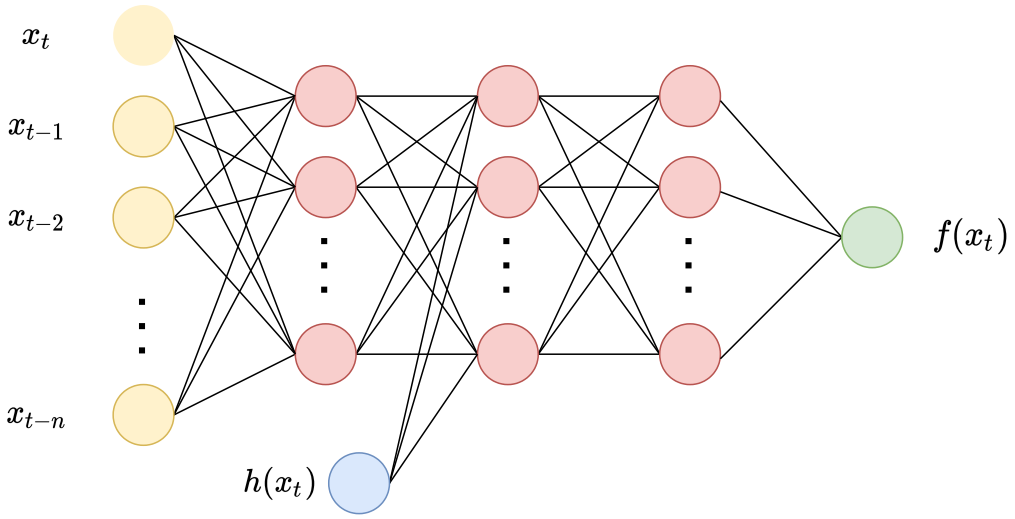


Figure 3.1.1: The framework of a PGNN with a injection at layer 2

Unlike other approaches such as regularization based on governing equations, the PGNN framework also incorporates the physics of the problem during the prediction stage rather than only during training.

$$x_{n+1} = x_n + \nabla t \cdot f(t_n, x_n) \tag{3.1.1}$$

Real-world tasks such as financial market prediction and weather forecasting often involve a time component. Time series forecasting is an important application of machine learning, and several methods have been developed for this, such as recurrent neural networks (RNN). RNNs are a generalization of the feedforward neural networks (FNN) discussed in Chapter 2, but with an internal memory making them capable of tasks such as speech recognition [33]. However, for this paper, a FNN was used with the sliding window technique to input a time series. The window slides over the data, capturing snippets of the data around the current time value, as illustrated in Figure 3.1.2. Choosing the window size is not straightforward and depends on the task, but previous studies [34] show that large sliding windows do not necessarily yield better results and that a window size of 5 might be sufficient.

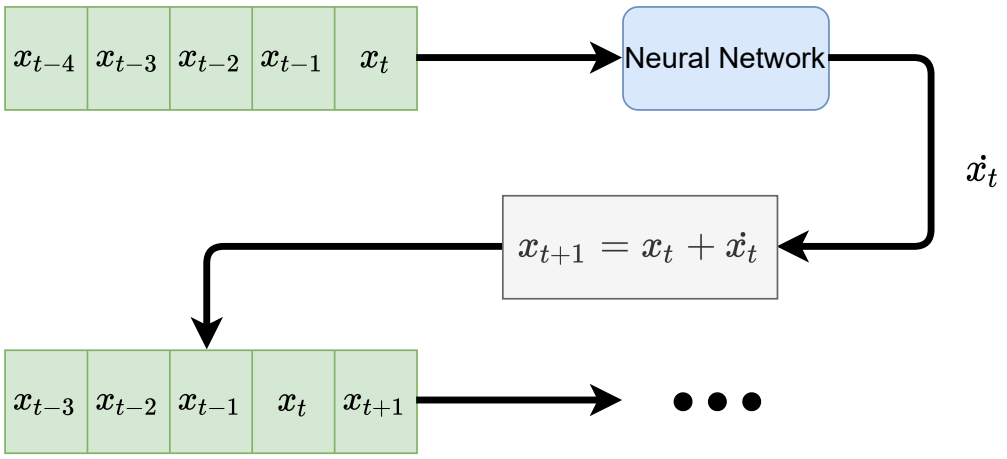


Figure 3.1.2: The sliding window technique with a window size of 5 during the prediction phase.

3.2 Data generation process

Data processing can often be the most challenging part of machine learning as bad input produces bad output. The curse of dimensionality was a limiting factor when choosing the dynamical systems since a high-dimensional feature space would require substantial data and training time. Also, the data should be stationary, i.e., the statistical properties of the process do not change over time, unlike stock prices and weather data. To reduce time dependency, non-

stationary data can be transformed into stationary data by *differencing*. Differencing computes the differences between the consecutive observations, reducing any trend and seasonality as shown below:

$$\dot{x_n} = \frac{x_n - x_{n-1}}{t_n - t_{n-1}}$$

For generating the data, the Python library SciPy [35] offers functions to solve a system of ODEs. It uses the Runge-Kutta 4(5) (RK45) method to solve the systems accurately enough for these experiments to deem the error negligible. The RK45 method produces a time series $[\boldsymbol{x}, \boldsymbol{t}]$, where x_i corresponds to the value of the system at time step t_i.

After simulating the systems, the time series from the RK45 method was divided into snippets of window size 5. These snippets would be the input for the neural networks, while the output would be the difference between the next element and the latest element, as shown in Table 3.2.1.

Input	Target
$X_0 = \begin{bmatrix} x_0 & x_1 & \dots & x_n \\ t_0 & t_1 & \dots & t_n \end{bmatrix}$	$f_0(x,t) = \frac{(x_{n+1} - x_n)}{(t_{n+1} - t_n)}$
$X_1 = \begin{bmatrix} x_1 & x_2 & \dots & x_{n+1} \\ t_1 & t_2 & \dots & t_{n+1} \end{bmatrix}$	$f_1(x,t) = \frac{(x_{n+2} - x_{n+1})}{(t_{n+2} - t_{n+1})}$
$\vdots$	$\vdots$

Table 3.2.1: Training data arrangement

3.2.1 Lotka–Volterra/Experiment 1

The first experiment tested is the Lotka-Volterra Equation 3.2.1, also known as the predator-prey equations [36]. It is a relatively simple system with a pair of first-order nonlinear differential equations describing the relationship between two species interacting, where x is prey, and y is a predator. An example is shown in Figure 3.2.1. As both equations are dependent on xy, this experiment

will compare the performance between a normal NN with a PGNN injected with xy at various layers.

$$\begin{aligned} \dot{x} &= \alpha_1 x - \alpha_2 xy \\ \dot{y} &= \alpha_3 xy - \alpha_4 y \end{aligned} \tag{3.2.1}$$

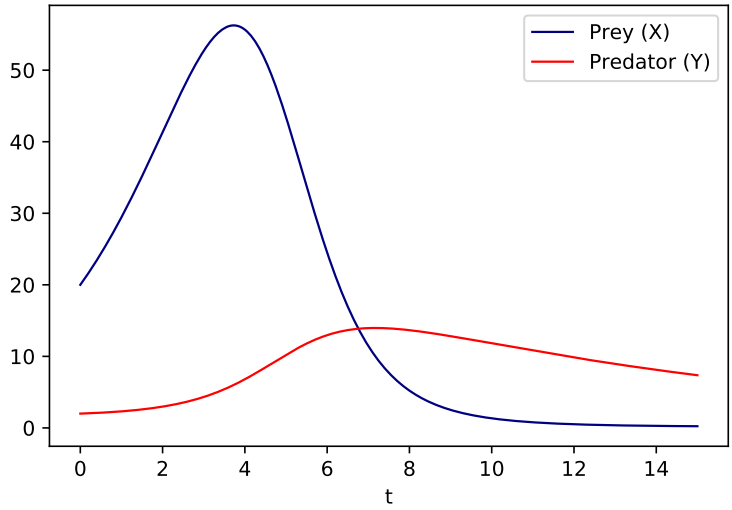


Figure 3.2.1: Lotka-Volterra system with $\alpha_1 = 0.6, \alpha_2 = 0.1, \alpha_3 = 0.1, \alpha_4 = 0.01$ and initial values $x_0 = 20, y_0 = 2$

For both training and testing, the parameters were $\alpha_1 = 0.6$, $\alpha_2 = 0.1$, $\alpha_3 = 0.1$, and $\alpha_4 = 0.01$ Two sets of data were generated for this experiment. First was when the initial values of the test data were inside the training data range, also known as *interpolation*. The other one predicted with values outside of the training data range, known as *extrapolation*, a relatively common problem in machine learning context [37]. The simulations would run for 15 seconds with a max time step of 0.05s, using the initial values found in Table 3.2.2.

3.2.2 Experiment 2/Duffing

The second experiment is the Duffing oscillator in Equation 3.2.2, which is a nonlinear second-order differential equation used to model an oscillator with both linear and nonlinear damping [38]. An example can be seen in Figure 3.2.2.

	x_0	y_0
Training	7	2
	⋮	⋮
	19	2
	21	2
	⋮	⋮
	30	2
Testing	20	2

(a) Interpolation

	x_0	y_0
Training	7	2
	⋮	⋮
	30	2
Testing	40	2

(b) Extrapolation

Table 3.2.2: Initial values for the Lotka-Volterra experiment

$$\ddot{x} = \beta_1 \, cos(\beta_2 t) - \beta_3 \dot{x} - \beta_4 x - \beta_5 x^3 \tag{3.2.2}$$

For this experiment, the main purpose was to investigate how changing the injected input would affect the performance. When generating the data, the system was simulated 25 times for 25 seconds with a max timestep of $0.05s$. Here, the initial values x_0 and $\dot{x}_0$ were randomly generated numbers between -1 and 1. One of these samples were chosen as the test sample.

3.2.3 Experiment 3/Chaotic systems

The Lotka-Volterra and Duffing equations are relatively stable systems with few parameters. To test the robustness of the PGNN framework, the following experiments would test the PGNN framework on some selected chaotic systems with more dimensions and parameters. The first system is the Lorenz system [39], a chaotic system of the ODEs in Equation 3.2.3.

$$\begin{aligned} \dot{x} &= \gamma_1(y - x) \\ \dot{y} &= x(\gamma_2 - z) - y \\ \dot{z} &= xy - \gamma_3 z \end{aligned} \tag{3.2.3}$$

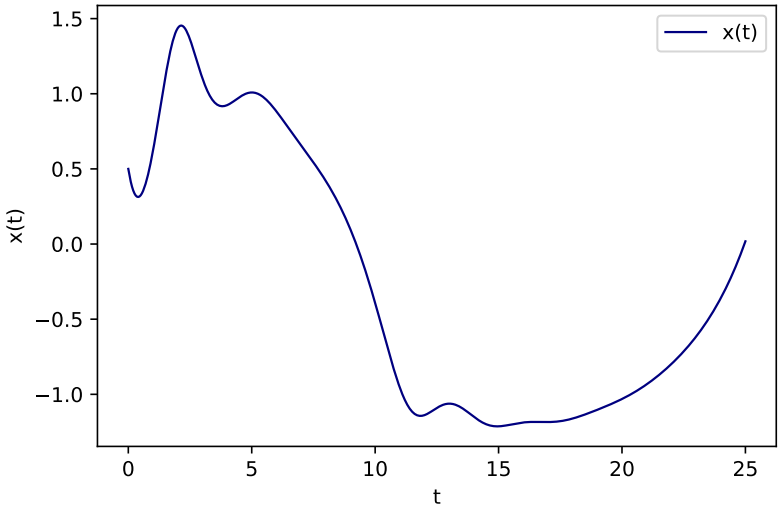


Figure 3.2.2: Duffing system with $\beta_1 = 2.3, \beta_2 = 0.2, \beta_3 = 1, \beta_4 = 0.5, \beta_5 = 1$ and initial values $x_0 = 0.5, x_0' = -1$

The next chaotic system is the Hindmarsh–Rose model [40]. It is a system of three nonlinear ordinary differential equations representing the bursting behavior of the membrane potential observed in experiments made with a single biological neuron. The system equations are in Equation 3.2.4.

$$\begin{aligned} \dot{x} &= y - \gamma_1 x^3 + \gamma_2 x^2 - z + \gamma_3 \\ \dot{y} &= \gamma_4 - \gamma_5 x^2 - y \\ \dot{z} &= \gamma_6[\gamma_7(x - \gamma_8) - z] \end{aligned} \tag{3.2.4}$$

The last system is the Willamowski-Rössler model [41] and is the most advanced one with 10 parameters. Its ODEs are in Equation 3.2.5, which represents chaos in chemical reactions. An example of all the systems can be seen in Figure 3.2.3 and their parameters in Table 3.2.3. While testing robustness was mainly the focus of this experiment, there was an ambition to find various patterns regarding the PGNN.

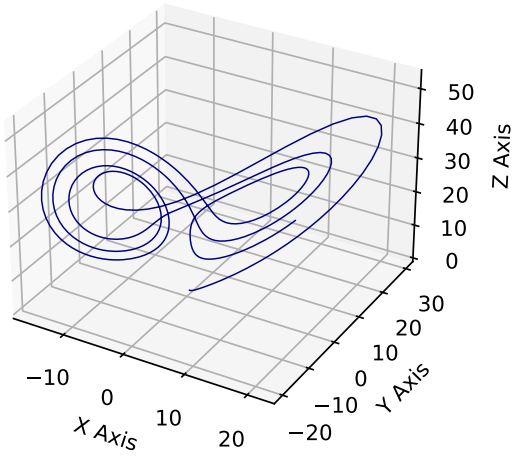
Parameter	Lorenz	Hindmarsh-Rose	Williamowski-Rössler
γ_1	10	1	30
γ_2	28	3	0.25
γ_3	$\frac{8}{3}$	5	1.0
γ_4	N/A	1	0.0001
γ_5	N/A	5	1.0
γ_6	N/A	0.001	0.001
γ_7	N/A	4	10
γ_8	N/A	-1.6	0.001
γ_9	N/A	N/A	16.5
γ_{10}	N/A	N/A	0.5

Table 3.2.4: Parameters and their values for the Lorenz, Hindmarsh-Rose, and Williamowski-Rössler models.

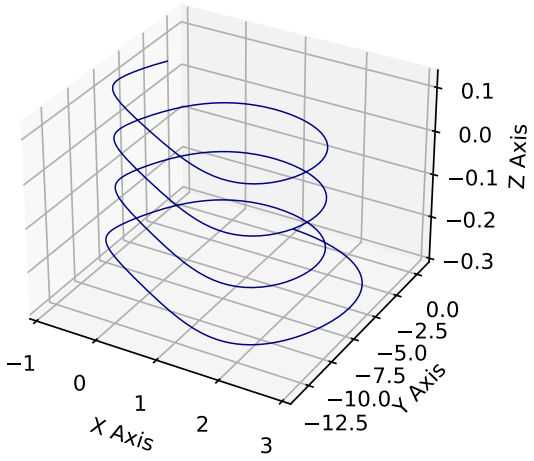
$$\begin{aligned} \dot{x} &= \gamma_1 x - \gamma_2 x^2 - \gamma_3 xy + \gamma_4 y^2 - \gamma_5 xz + \gamma_6 \\ \dot{y} &= \gamma_3 xy - \gamma_4 y^2 - \gamma_7 y + \gamma_8 \\ \dot{z} &= -\gamma_5 xz + \gamma_6 + \gamma_9 z - \gamma_{10} z^2 \end{aligned} \tag{3.2.5}$$

3.3 Hyperparameters

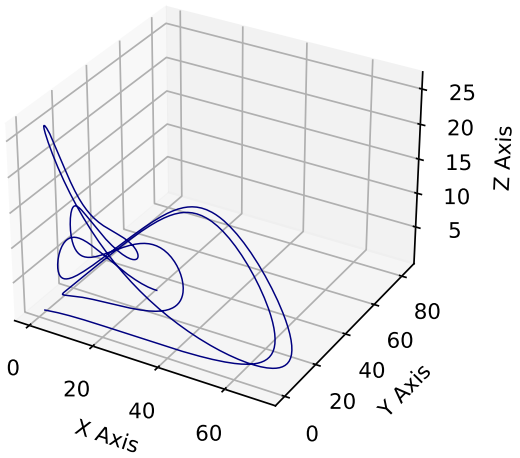
Since these experiments' focus was to investigate the effect of the PGNN, the hyperparameters would remain mostly the same for all networks to mitigate their effect and keep the experiments under controlled conditions. The non-deterministic nature of NN training made it essential to choose hyperparameters that would yield as stable results as possible. In earlier experiments, some outliers would affect the results to such as degree that the results were not representative. A network's size can affect the stability and ability to approximate the function to a large degree. For experiments 1 and 2, the networks had three hidden layers with 16, 32, and 16 neurons, which should be an acceptable balance between stability and a challenge for the networks to approximate the systems. As the systems in experiment 3 had more dimensions and parameters,



(3a) Example of the Lorenz system with initial values $x_0 = 0, y_0 = 1, z_0 = 0$



(3b) Example of the Hindmarsh–Rose model with initial values $x_0 = 1, y_0 = 0.5, z_0 = -0.3$



(3c) Example of the Williamowski-Rössler model with initial values $x_0 = 0, y_0 = 1, z_0 = 0$

Figure 3.2.3: Example plots of the chaotic systems tested.

the networks had three layers with 32, 64, and 32 neurons.

Deciding the numbers of epochs was based on the model loss plots during training for each system individually. Figure 3.3.1 shows the training loss for the Lotka-Volterra system for 300 epochs. However, after 50 epochs, the model's loss changed so slowly that the computational time would outweigh the loss reduction. So even though training could minimize the loss even further, it had reached a satisfactory level, and the training stopped early to reduce time and avoid overfitting. This was done for every experiment. Another solution to avoid overfitting would be to expand the training data, as more data is often better.

Figure 3.3.1: Training loss for Lotka-Volterra over 300 epochs

Instead of having a fixed learning rate, the networks utilize the Adam optimizer, a stochastic gradient descent method that is computationally efficient and has little memory requirement [42]. It maintains separate learning rates for each network parameter and adapts them as learning unfolds, which is convenient for problems with large amounts of data/parameters. Some of the hyperparameters are in table 3.3.1.

Hyperparameter	Value
Activation function	ReLU
Batch size	32
Validation split	0.2
Loss function	MSE
Learning rate	0.001
Adam optimizer	$\beta_1 = 0.9, \beta_2 = 0.999$

Table 3.3.1: Hyperparameters kept constant throughout the experiments.

3.4 Hardware/Software Specification

One of the most significant factors in the viability of ML algorithms is recent advances in computational hardware. However, not all modern computers can carry out such tasks efficiently. The GPU, arguably the most essential tool, is not always present in a computer. Unfortunately for this thesis, all experiments ran on an Intel Core i7-8550U CPU. As some run times could surpass 20 hours, better hardware would have enabled more comprehensive experiments.

For setting up the neural network architecture, the ML library Keras [43] version 2.4.3 was used. Keras features the *functional API* that allows the creation of more flexible models with, e.g., multiple inputs at different layers, making it suitable for this experiment. Data visualization was done by the python libraries *matplotlib* [44] and *seaborn* [45]. Everything was done in Python 3.8.5.

4 Results and Discussions

This chapter presents the results from each experiment introduced in Chapter 3. Each experiment undergoes various circumstances to test the robustness of the PGNN framework. Experiments 1 and 2 test the Lotkav-Volterra system and the Duffing equation, respectively. Experiment 3 focuses on three chaotic systems to check performance on more complex systems and examine any patterns.

4.1 Lotka-Volterra

As the Lotka-Volterra system in Equation 3.2.1 is relatively simple, injecting xy was expected to have a substantial impact on the results. Already in the training phase, there was a clear difference between the PGNN and the normal NN. Figure 4.1.1 shows the average training loss over 100 initializations for a normal NN and PGNN with an injection at different layers. While most PGNNs trained better than the normal NN, the PGNN with an injection at the output layer had almost equal training loss as the normal NN. One cause might be the backpropagation algorithm that needs more layers to calculate the gradient of the loss. The output layer uses a linear activation function that returns the weighted sum of the input without changing it, possibly making it harder for the network to supplement the injection.

Even though there was little difference between the network injected at layers 1, 2, and 3, injection at layer 2 had slightly lower training loss. It is unclear where it is best to inject the guided input and why, but a slight trend hints towards the earlier layers. Looking at the predictions, shown in Figure 4.1.2, layer 2 was also the best. Therefore, future plots where PGNN and normal NN are compared will be PGNN injected at layer 2.

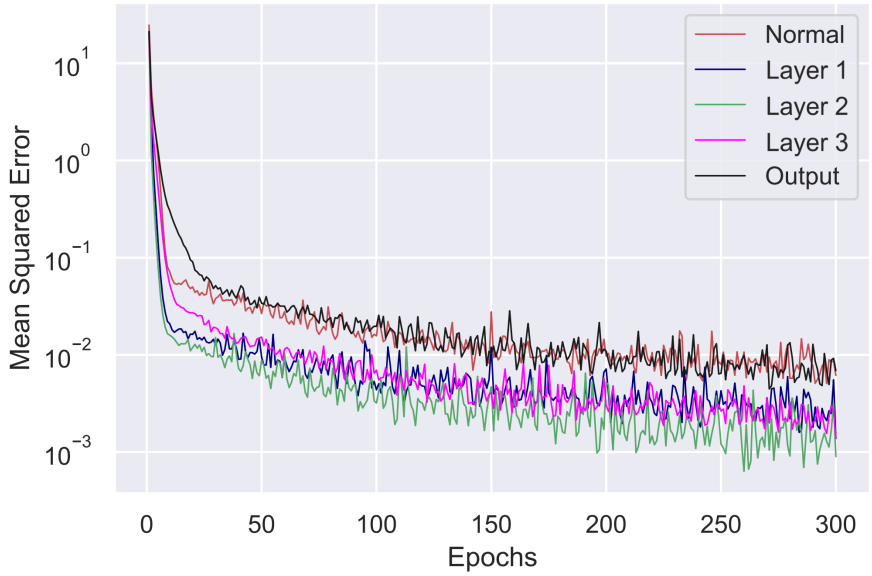
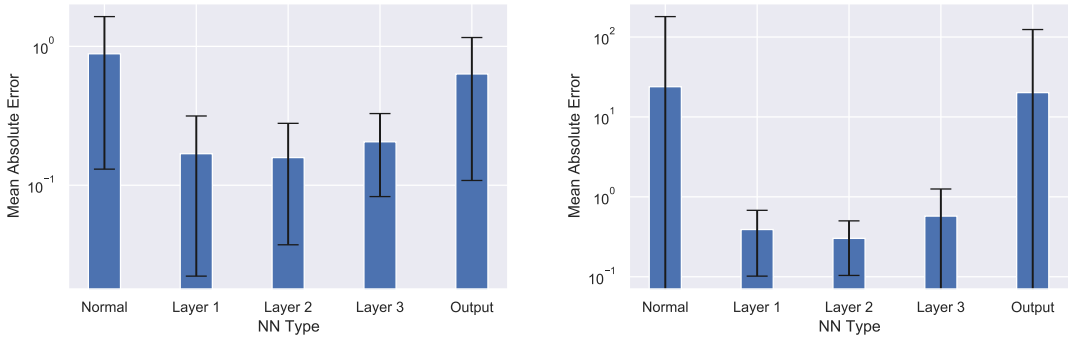


Figure 4.1.1: Average training loss for the Lotka-Volterra system for 300 epochs. Each type had 100 initializations each.

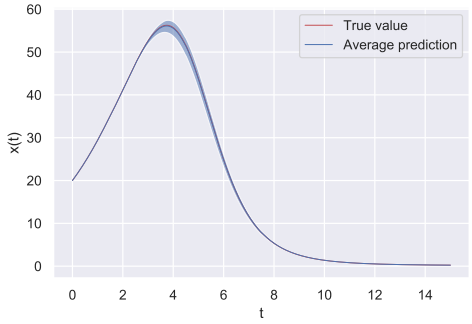


(2a) Interpolation **(2b)** Extrapolation

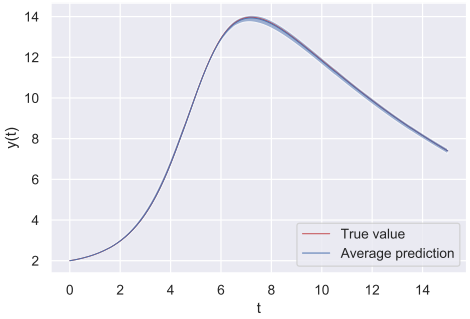
Figure 4.1.2: Average prediction error for the Lotka-Volterra system during interpolation and extrapolation. Average is over 100 networks trained for 50 epochs.

Two sets of data were tested, interpolation and extrapolation. For the interpolation experiment, injecting xy mostly impacted keeping the network's predictions more stable. Figure 4.1.3 shows the true value and the average predictions with a 95% confidence interval of the 100 initializations. The average predictions are only slightly better with injection, as both types approximate

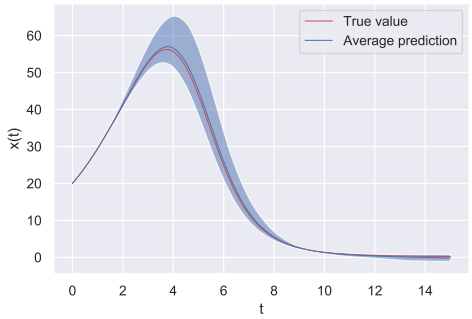
the functions sufficiently. The biggest difference is how the injection decreases the variance and keeps the predictions more stable.



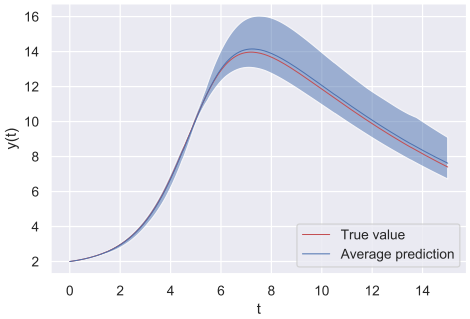
(3a) Injection of xy at layer 2



(3b) Injection of xy at layer 2



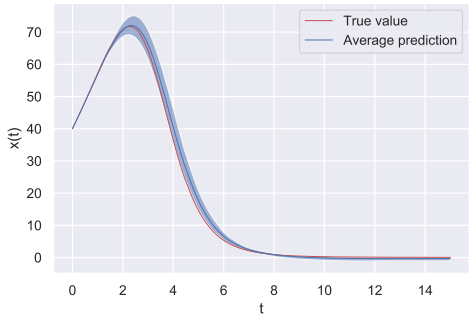
(3c) No injection



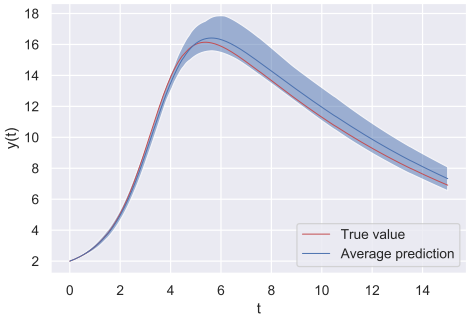
(3d) No injection

Figure 4.1.3: Average predictions of the Lotka-Volterra equation over 100 networks (Interpolation) with a 95% confidence interval.

For the extrapolation part, the training data went from $x_0 \in [7, 30]$ while the test data had $x_0 = 40$, which was well outside the training data. Here, the difference between PGNN and the normal NN was even greater, as shown in Figure 4.1.4. With the injection, the average predictions had much better accuracy, especially towards the end. Also, the confidence interval was much narrower with the injection. It seems like the normal NN had trouble with overshooting when there were sudden changes and diverging towards the end, while the injection helped constrain the PGNN to a more representative result.

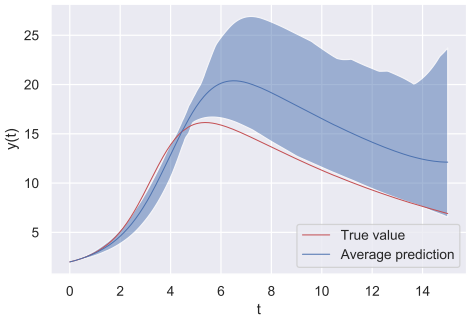


(4a) Injection of xy at layer 2



(4b) Injection of xy at layer 2

(4c) No injection



(4d) No injection

Figure 4.1.4: Average predictions of the Lotka-Volterra equation over 100 networks (Extrapolation) with a 95% confidence interval

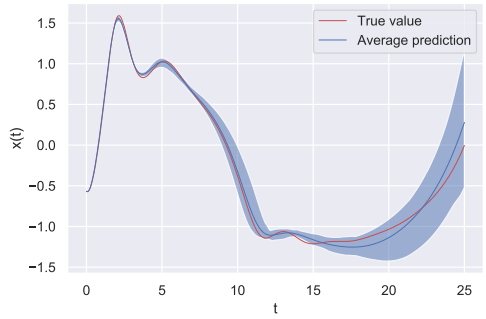
Overall, there is a clear trend that injecting xy helped the PGNN during training and predictions. While not distinct which layer gave the best results, injecting at the earlier and middle layers was the best option. Even with the small network size, a normal NN can approximate the system efficiently, especially in the interpolation case. It starts to struggle for the extrapolation case, and it is here that the PGNN outperforms the normal NN noticeably. Increasing the size of the normal NN, either amount neurons or layers, would most likely improve its performance; however, deeper networks are harder to train [46].

4.2 /Duffing

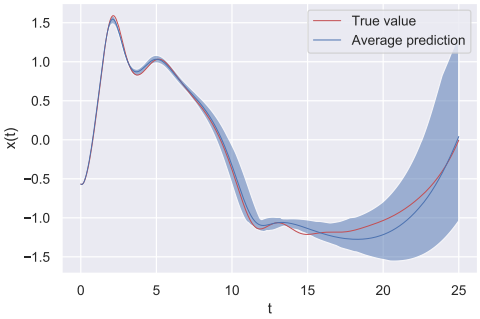
The Duffing equation is a second-order differential equation with many different terms. Part of this experiment was investigating how injecting different functions would affect the results. Figure 4.2.1 shows predictions of the Duffing equation for a normal NN and PGNN with different injections at layer 2. Comparing the different plots shows similar results to the Lotka-Volterra system. Average prediction only slightly improved with injections, but the injections contracted the confidence interval and made the predictions more stable. There is also a notable difference between the functions, as injecting x^3 made considerable improvements while injecting $cos(\beta_2 t)$ barely made any improvements. Injecting both simultaneously makes no difference to just injecting x^3, clearly making it the essential part to inject.

As there was a definite difference between injecting $cos(\beta_2 t)$ and x^3, it is interesting to investigate why. Figure 4.2.2 shows how two NN with the same parameters and data size predicted the functions $cos(x)$ and x^3. The results show that NNs require much less training to approximate $cos(x)$ than x^3 at a sufficient level. Therefore, the PGNN must most likely be injected with functions NNs approximate poorly, such as x^3, to be successful. Otherwise, it seems to add little to no effect, questioning its practicality in some cases.

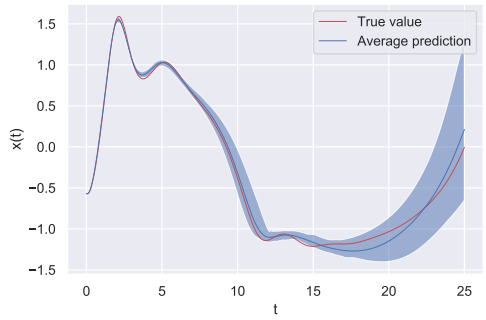
In real-world applications, noise is inevitable as physical sensors are limited and can not register the values flawlessly. Sometimes noisy data result in lower accuracy and poor prediction results [47], but can also be added to the training data to aid generalization and fault tolerance [48, 49]. To examine PGNNs noise sensitivity, one test added Gaussian noise with a mean of 0 and a standard deviation of 0.05 to the Duffing data. Figure 4.2.3 shows the prediction differences between a normal NN and a PGNN injected with x^3, trained on noisy data. The most significant difference was how the confidence interval of the PGNN expanded with noise, but its average predictions remained the same. It could suggest PGNNs being sensitive to noise, depending on the injected function. x^3 grows exponentially, and slight variations in x could critically impact the network. $cos(\beta_2 t)$ is not dependent on x and is most likely why noise barely affected it.



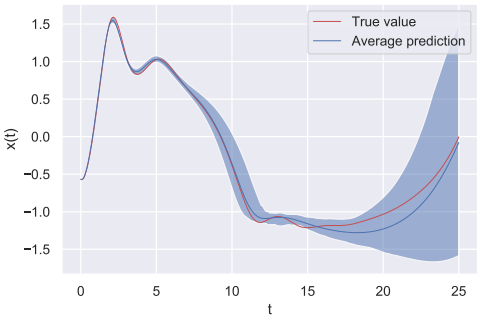
(1a) Injection of x^3



(1b) Injection of $cos(\beta_2 t)$



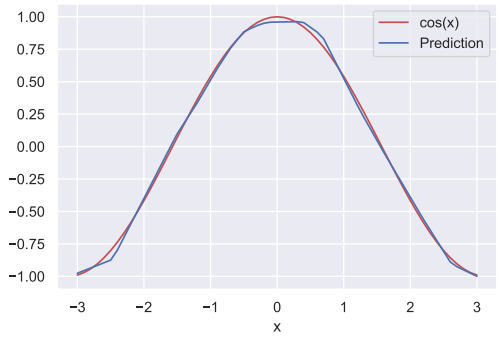
(1c) Injection of x^3 and $cos(\beta_2 t)$



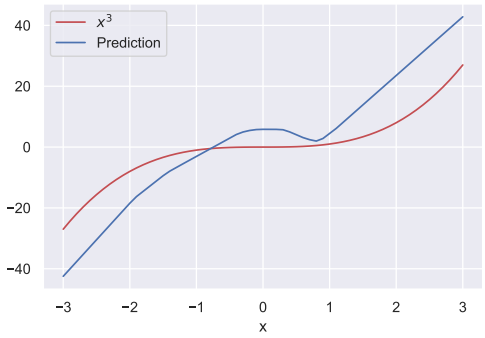
(1d) No injection

Figure 4.2.1: Average predictions for Duffing equation over 100 networks with 95% confidence interval.

The last test on the Duffing equation was how increasing the network size affected the results. As stated before, increasing the network size can aid in approximating functions, but make them harder to train and interpret. Figure 4.2.4 shows how the normal NN and PGNN performed at different network sizes, from 16, 32, and 16 neurons per layer to 64, 128, and 64 neurons per layer. It shows that the PGNN can achieve the same results as a normal NN with a lot smaller size, probably because the network needs to approximate fewer parts of the function. Decreasing the network size makes it easier to analyze how and why the network behaves as it does, and is a big step towards increasing NNs interpretability.

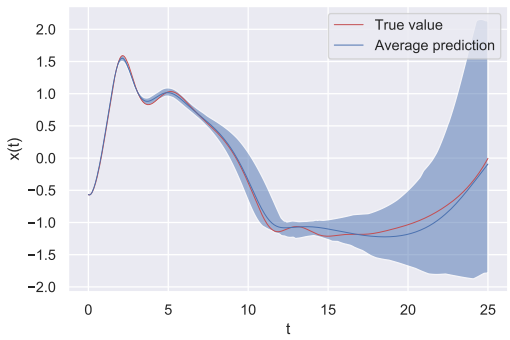


(2a) NN predicting $cos(x)$

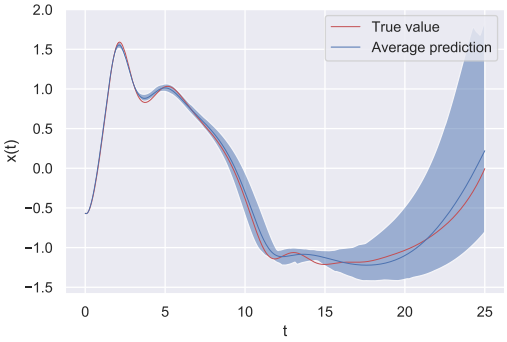


(2b) NN predicting x^3

Figure 4.2.2: Comparison of NN with same parameters predicting two different functions.



(3a) Without injection

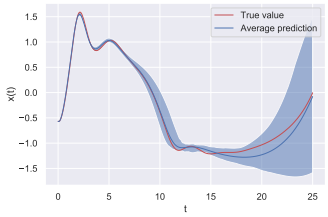


(3b) With x^3 injected

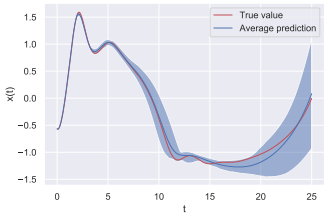
Figure 4.2.3: Average predictions of Duffing with added noise to training data over 100 networks.

4.3 Chaotic systems

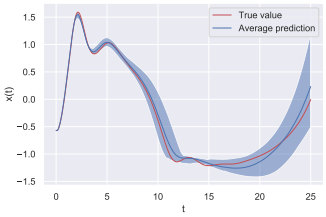
Three different chaotic systems were tested in the final experiment, each with three dimensions. For the Lorenz system, the networks trained for 20 epochs and had 50 initializations instead of 100 due to time-saving. For this system, there was no apparent difference between the PGNN and normal NN, as shown in Figure 4.3.1. Both networks were able to approximate the functions suitably with no difficulties. Perhaps the system was relatively easy to approximate for a NN, or they had an excessive amount of training. The results were similar for all three dimensions, so it only seemed necessary to plot one dimension.



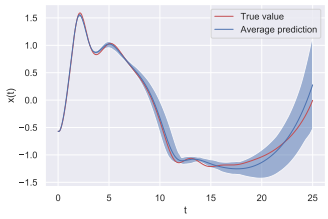
(4a) No injection with hidden layers of size 16, 32 and 16.



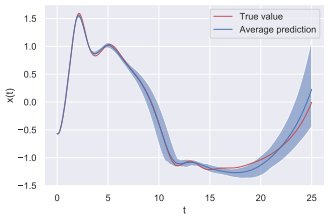
(4b) No injection with hidden layers of size 32, 64 and 32.



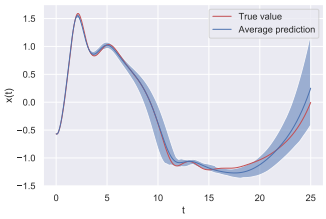
(4c) No injection with hidden layers of size 64, 128 and 64.



(4d) Injecting x^3 with hidden layers of size 16, 32 and 16.



(4e) Injecting x^3 with hidden layers of size 32, 64 and 32.

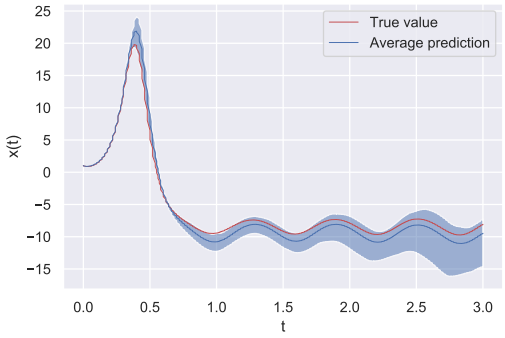


(4f) Injecting x^3 with hidden layers of size 64, 128 and 64.

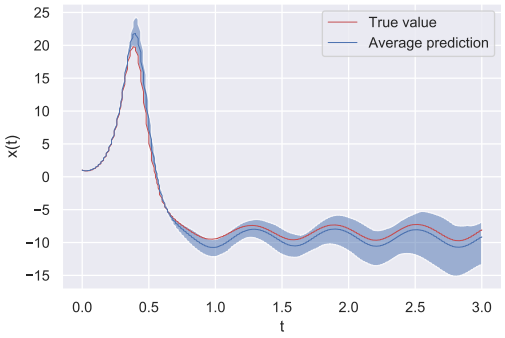
Figure 4.2.4: Average predictions of the Duffing system with increasing network sizes for 100 networks each. Top row is normal NN, while bottom row is PGNN injected with x^3.

For the Hindmarsh-Rose model, the average predictions were almost equal, with the only difference in the confidence interval. Similar to the Lotka-Volterra and Duffing experiments, the PGNN's confidence interval is slightly slimmer than the normal NN. On this system, the PGNN was injected with x^3, which is hard to approximate for NNs, as we know from earlier. The difference is minuscule, but there was little to improve as the normal NN predicted very accurately.

The Willamowski-Rössler model was the most complex system, which both networks struggled to approximate. Unfortunately, given the nondeterministic nature of NN, some of the predictions of the normal NN were unrepresentative. Figure 4.3.3 show the predictions for the Willamowski-Rössler model, and it is clear that some of the simulations skewed the normal NNs' results, causing them to diverge considerably. However, the PGNN did not experience this, indi-

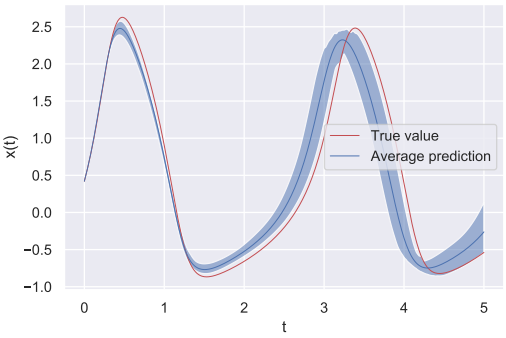


(1a) Without injection

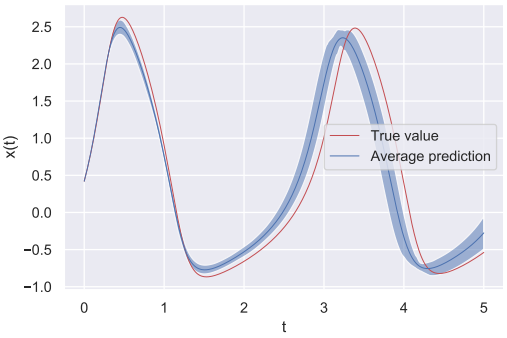


(1b) With injection of xy

Figure 4.3.1: Average predictions of the Lorenz system with a 95% confidence interval over 50 networks.



(2a) Without injection

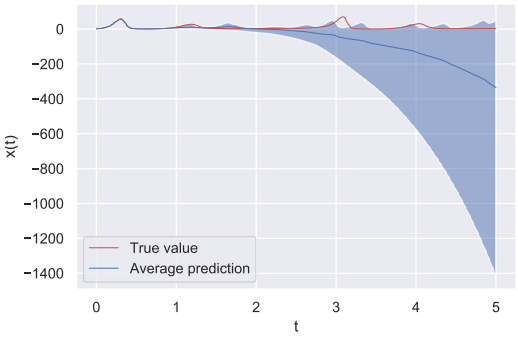


(2b) With injection of x^3

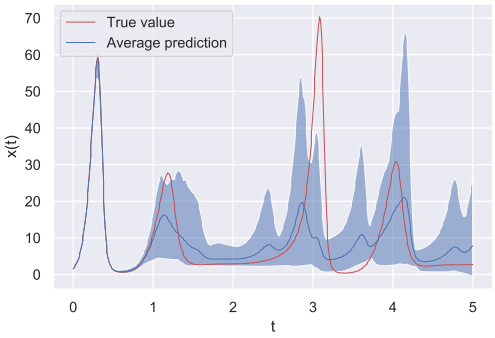
Figure 4.3.2: Average prediction the Hindmarsh-Rose model with a 95% confidence interval over 50 networks.

cating an ability to make more stable predictions over more prolonged periods.

To investigate PGNNs' ability to make more stable predictions over more prolonged periods, some systems were simulated for a longer time. In Figure 4.3.4, the Lorenz system was simulated for 15 seconds. Unlike the shorter simulations where the normal NN and PGNN performed evenly, the PGNN had more stable predictions in this situation. While both networks end up diverging, the normal NN diverges quicker and to a much greater extent than the PGNN. This was also



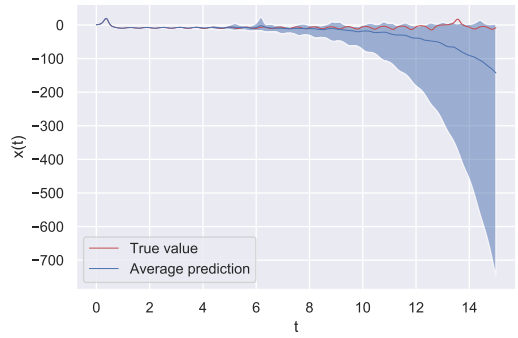
(3a) Without injection



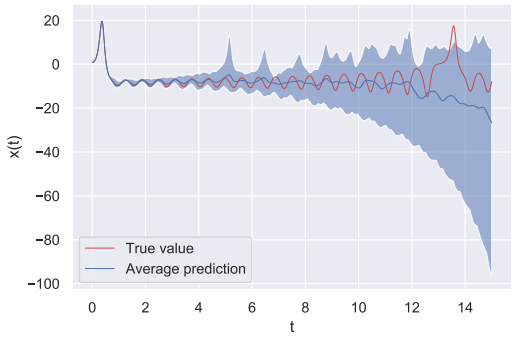
(3b) With injection of xy

Figure 4.3.3: Average prediction of the Willamowski-Rössler model with a 95% confidence interval over 50 networks.

the case for the Hindmarsh-Rose system with extended simulations, just not to the same degree as the Lorenz system. Therefore, it seems likely that PGNNs can make more stable predictions compared to normal NN when simulated over longer periods.



(4a) Without injection



(4b) With injection of xy

Figure 4.3.4: Average prediction of the Lorenz system simulated for 15 seconds, with a 96% confidence interval over 50 networks

5 Conclusion and future work

5.1 Conclusion

From the results, there is evidence to support that the PGNN framework **can** enhance accuracy and stability. Whenever the injected information was a vital part of the system, like xy in the Lotka-Volterra equation, the PGNN outperformed the normal NN in many circumstances. In particular, PGNN could predict with much greater accuracy and stability in the extrapolation case where the test data was outside the training data range. Real-world scenarios will often have incomplete and scarce data, so the generalizability of the PGNN could be valuable in such cases.

Injecting different types of functions had a noticeable impact on the results in some cases. There was a substantial difference between injecting x^3 and $cos(\beta_2 t)$ in the Duffing equation, where x^3 drastically improved the predictions while there was little difference when injecting $cos(\beta_2 t)$. A big reason for that is the required training a NN needs to approximate each function, as x^3 is much harder for a NN to approximate than $cos(x)$. However, injecting x^3 in the Hindmarsh-Rose model did not make a prominent difference. For most of the larger systems, the injection made no apparent effect until some time had passed. The normal NNs tended to diverge during prediction, growing exponentially with time. On the other hand, the PGNN avoided divergence much more than a normal NN, even when the injected function was relatively simple.

In the cases where the PGNN performed better, the complexity of the networks could be reduced by decreasing the size. For a normal NN to perform equivalent to the PGNN on the Duffing equation, the number of neurons doubled.

Reducing the size of the networks increases their interpretability and makes it easier to conduct analyses to draw inferences about the network's prediction and learning ability.

Despite encouraging results in both prediction and interpretability, there are still several uncertainties regarding the PGNN. The results have been widely disparate, and it is unclear what defines a problem where PGNN will be helpful. This thesis has undergone various experiments and could give further research a more guided direction. Until now, the PGNN has only performed on systems of ODEs, but has potential in several other fields.

5.2 Future Work

5.2.1 More advanced system

Up until now, the PGNN tested on relatively simple systems with generated data. If the PGNN framework is to perform on a real-world application, it would need testing on a more realistic situation, where the data is from observations like sensor reading. As we saw in the results section, the PGNN might be sensitive to noise depending on the injected function, so a comprehensive investigation is needed to see how it would perform on real-world data. Also, generated data has little to no uncertainties as the injected equations are part of the systems. So in a realistic system, the injected functions might be certain in the mathematical equations, but inaccurate in reality.

5.2.2 More testing

One crucial issue not addressed enough in this thesis is whether there are a priori indicators in the system that indicate whether PGNN will be applicable. There needs to be further comprehensive testing of the PGNN on a much larger number of systems to look for more patterns. Looking at the results, the PGNN was more helpful when the injected information was a substantial part of the system, like the Lotka-Volterra equations, but for the chaotic systems, the PGNN was less beneficial. More testing could determine the type and amount of information that must be injected for a noticeable effect.

5.2.3 Other network types

The proposed framework is quite adaptable and could work for many other machine learning algorithms. Without going too far away, the framework could be applied to a recurrent neural network. As mentioned in Chapter 3, these networks have internal memory nodes capable of processing long sequences of inputs. Modular neural networks have multiple networks that function independently and perform sub-tasks. Here, the networks could be injected with information only relevant to their sub-task.

Bibliography

[1] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[2] Amina Adadi and Mohammed Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160, 2018.

[3] Anuj Karpatne, William Watkins, Jordan Read, and Vipin Kumar. Physics-guided neural networks (pgnn): An application in lake temperature modeling, 2018.

[4] Emmanuel de Bezenac, Arthur Pajot, and Patrick Gallinari. Deep learning for physical processes: Incorporating prior scientific knowledge, 2018.

[5] E. Kharazmi, Z. Zhang, and G. E. Karniadakis. Variational physics-informed neural networks for solving partial differential equations, 2019.

[6] David J. Lucia, Philip S. Beran, and Walter A. Silva. Reduced-order modeling: new approaches for computational physics. *Progress in Aerospace Sciences*, 40(1):51–117, 2004.

[7] David Amsallem. Reduced order modelling. `https://web.stanford.edu/group/frg/active_research_themes/reducedmodel.html`.

[8] Mohammad Abid Bazaz, Mashuq un Nabi, and S. Janardhanan. A review of parametric model order reduction techniques. In *2012 IEEE International Conference on Signal Processing, Computing and Control*, pages 1–6, 2012.

[9] Kunihiko Taira, Steven L. Brunton, Scott T. M. Dawson, Clarence W. Rowley, Tim Colonius, Beverley J. McKeon, Oliver T. Schmidt, Stanislav Gordeyev, Vassilios Theofilis, and Lawrence S. Ukeiley. Modal analysis of fluid flows: An overview. *AIAA Journal*, 58(11):AU9–AU9, 2020.

[10] Eivind Fonn, Harald van Brummelen, Trond Kvamsdal, and Adil Rasheed. Fast divergence-conforming reduced basis methods for steady navier–stokes flow. *Computer Methods in Applied Mechanics and Engineering*, 346:486–512, 2019.

[11] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

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

[13] Maziar Raissi, Zhicheng Wang, Michael S. Triantafyllou, and George Em Karniadakis. Deep learning of vortex-induced vibrations. *Journal of Fluid Mechanics*, 861:119–137, Dec 2018.

[14] U. Forssell and P. Lindskog. Combining semi-physical and neural network modeling: An example ofits usefulness. *IFAC Proceedings Volumes*, 30(11):767–770, 1997. IFAC Symposium on System Identification (SYSID'97), Kitakyushu, Fukuoka, Japan, 8-11 July 1997.

[15] Suraj Pawar, Omer San, Burak Aksoylu, Adil Rasheed, and Trond Kvamsdal. Physics guided machine learning using simplified theories. *Physics of Fluids*, 33(1):011701, 2021.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[17] M. I. Jordan and T. M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[18] Warren S Sarle. Stopped training and other remedies for overfitting. *Computing science and statistics*, pages 352–360, 1996.

[19] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[20] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel Nothman, Gilles Louppe, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python, 2018.

[21] Geoffrey E. Hinton. *A Practical Guide to Training Restricted Boltzmann Machines*, pages 599–619. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[22] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning, 2015.

[23] Marc Claesen, Jaak Simm, Dusan Popovic, Yves Moreau, and Bart De Moor. Easy hyperparameter search using optunity, 2014.

[24] Ilan Alon, Min Qi, and Robert J. Sadowski. Forecasting aggregate retail sales:: a comparison of artificial neural networks and traditional methods. *Journal of Retailing and Consumer Services*, 8(3):147–156, 2001.

[25] C. A. Mitrea, C. K. M. Lee, and Z. Wu. A comparison between neural networks and traditional forecasting methods: A case study. *International Journal of Engineering Business Management*, 1:11, 2009.

[26] Bayya Yegnanarayana. *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.

[27] Sagar Sharma. Activation functions in neural networks. *Towards Data Science*, 6, 2017.

[28] Forest Agostinelli, Matthew Hoffman, Peter Sadowski, and Pierre Baldi. Learning activation functions to improve deep neural networks, 2015.

[29] Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions, 2017.

[30] Scott M. Lundberg, Gabriel G. Erion, and Su-In Lee. Consistent individualized feature attribution for tree ensembles, 2019.

[31] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[32] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, 2015.

[33] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee, 2013.

[34] Ray J Frank, Neil Davey, and Stephen P Hunt. Time series prediction and neural networks. *Journal of intelligent and robotic systems*, 31(1):91–103, 2001.

[35] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Peterson, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[36] Eric W Weisstein. Lotka-volterra equations. `https://mathworld.wolfram.com/Lotka-VolterraEquations.html`. From MathWorld–A Wolfram Web Resource.

[37] Giles Hooker. *Diagnostics and extrapolation in machine learning*. stanford university, 2004.

[38] Eric W Weisstein. Duffing differential equation. `https://mathworld.wolfram.com/DuffingDifferentialEquation.html`. From MathWorld–A Wolfram Web Resource.

[39] Eric W Weisstein. Lorenz attractor. `https://mathworld.wolfram.com/LorenzAttractor.html`. From MathWorld–A Wolfram Web Resource.

[40] J. L. Hindmarsh, R. M. Rose, and Andrew Fielding Huxley. A model of neuronal bursting using three coupled first order differential equations. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 221(1222):87–102, 1984.

[41] J Güémez and MA Matías. Internal fluctuations in a model of chemical chaos. *Physical Review E*, 48(4):R2351, 1993.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[43] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[44] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

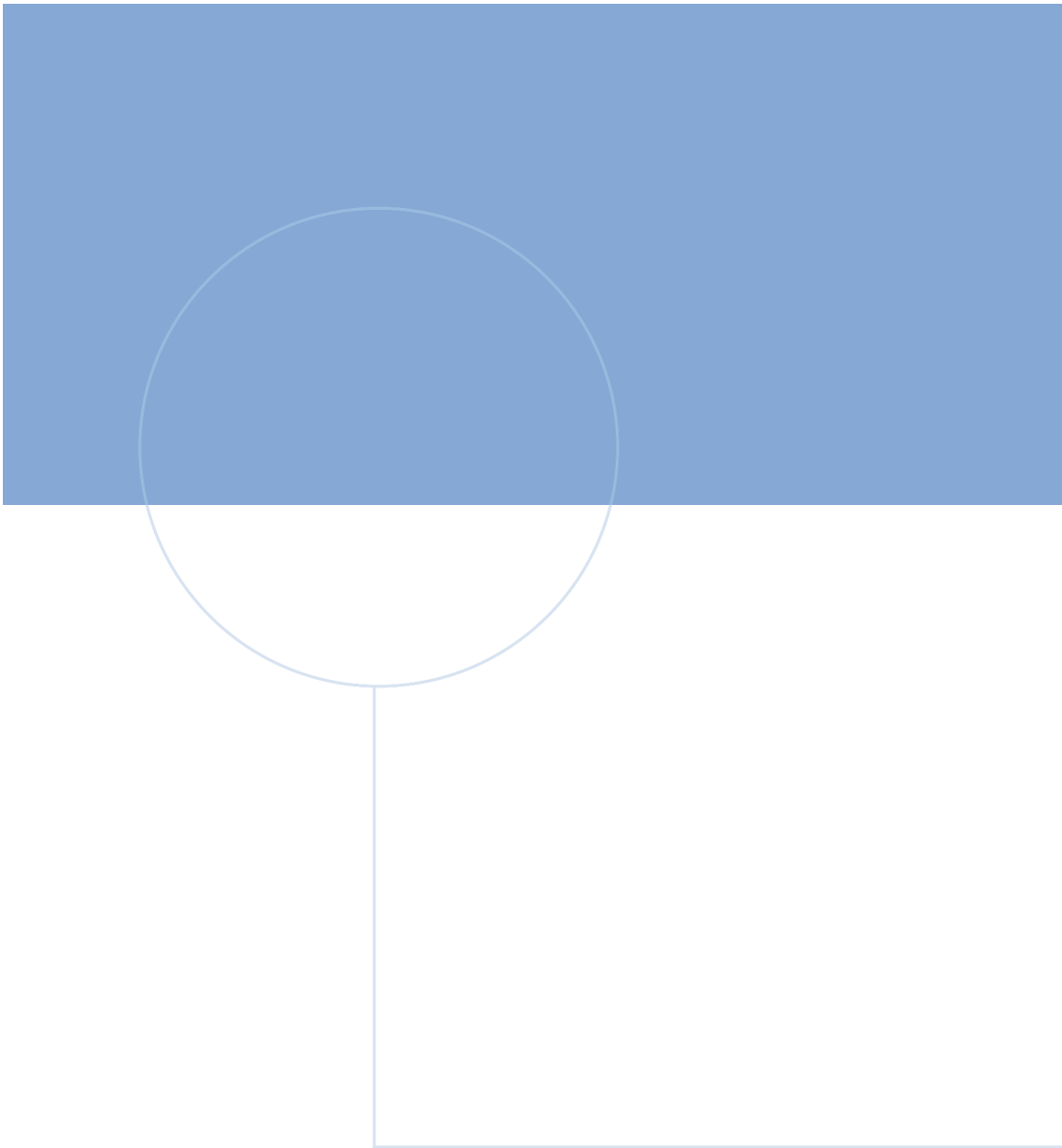
[45] Michael L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021.

[46] Simon S. Du, Jason D. Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai. Gradient descent finds global minima of deep neural networks, 2019.

[47] Shivani Gupta and Atul Gupta. Dealing with noise problem in machine learning data-sets: A systematic review. *Procedia Computer Science*, 161:466–474, 2019. The Fifth Information Systems International Conference, 23-24 July 2019, Surabaya, Indonesia.

[48] Russell Reed and Robert J MarksII. *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press, 1999.

[49] Christopher M Bishop et al. *Neural networks for pattern recognition*. Oxford university press, 1995.



NTNU

Kunnskap for en bedre verden