NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

FRANCISCO MANUEL DA IGREJA FIRMINO HENRIQUES

Degree in Computer Science and Engineering

MACHINE LEARNING FOR NONLINEAR INVERSE PROBLEMS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

MACHINE LEARNING FOR NONLINEAR INVERSE PROBLEMS

FRANCISCO MANUEL DA IGREJA FIRMINO HENRIQUES

Degree in Computer Science and Engineering

Adviser: Jorge Cruz
Assistant Professor, NOVA University Lisbon

Co-adviser: Ludwig Krippahl
Assistant Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

Machine Learning for Nonlinear Inverse Problems

To my parents.

Acknowledgements

In the first place, I would like to greatly thank my advisors, professor Jorge Cruz and Ludwig Krippahl, for their continuous and restless support throughout the development of this dissertation. Their guidance, insights, and knowledge of the subject matter steered me through this research. I would like to extend my gratitude to the Department of Computer Science of NOVA School of Science Technology, for all of the great learning structure they provided me over the past years.

I want to thank my family for all of their support, love, and encouragement shown during my academic path, especially to my mother Elisa and step-father Manuel who always gave me love and affection, always had my best interests in mind, letting me be who I wanted to be, and allowing me to pursue greater education.

To my friend Gabriel, my "partner in crime" for the last 5 years, thank you for all your help and patience, for being there when times were rough, and also to celebrate every achievement. And finally thank you to every friend made on this journey, in particular to Miguel, João, and Ana, you took me under your wings since day 1, helped me to integrate into this new institution, you were there when I doubted myself and helped me push through every obstacle in my way.

"If you only do what you can do, you will never be more than you are now." (Shifu)

Abstract

There have been multiple mathematical models presented by scientists that allow corroborating the behavior of complex systems. However, these models estimate values that can be measured but are unable to determine the parameters that caused such behaviors.

Inverse Problems aim at finding the parameters of a model, given by systems of equations, from noisy observations/measurements. These are typically ill-posed problems that may have no exact solutions, multiple solutions or unstable solutions. In this thesis, we will be restringing our work to nonlinear inverse problems that have an exact single solution but due to their complexity can not be computed analytically and a small uncertainty in the measurements may induce a large uncertainty in the solutions.

Our approach resorts to deep learning techniques in order to support reasoning for this family of non-linear inverse problems. In this thesis, we will employ the forward model to generate the dataset used to train the neural network which is going to be used as a regression model to approximate the desired inverse function.

This work will be applied to a research area widely used in climate change studies with potential applications in water quality monitoring, denominated Ocean Color. We aim to obtain a model that is capable of accurately estimating the concentration of active seawater compounds, from remote sensing measurements of the sea surface reflectance taking into consideration the impact of uncertainty on the sensor observations and the model approximations.

Keywords: Nonlinear inverse problems, Uncertainty representation and reasoning, Ocean color remote sensing, Deep learning, Neural network

Resumo

Houve múltiplos modelos matemáticos propostos por cientistas que permitem corroborar o comportamento de sistemas complexos. No entanto, estes modelos estimam valores que podem ser medidos porém são incapazes de determinar os parâmetros que causaram tais comportamentos.

Os problemas inversos visam encontrar os parâmetros de um modelo, dados por sistemas de equações, a partir de observações/medições ruidosas. Estes são tipicamente problemas *ill-posed* que podem não ter soluções exactas, ter múltiplas soluções ou soluções instáveis. Nesta tese, iremos restringir o nosso trabalho a problemas inversos não lineares que devido à sua complexidade não podem ser computados analiticamente e uma pequena incerteza nas medições pode induzir uma grande incerteza nas soluções.

A nossa abordagem recorre a técnicas de aprendizagem profunda com o intuito de arranjar soluções para resolver esta familia de problemas inversos não lineares. Nesta tese vamos empregar o modelo direto para gerar o conjunto de dados utilizado para treinar a rede neural que vai ser utilizado como modelo de regressão para aproximar a função inversa desejada.

Este trabalho será aplicado a uma área de investigação amplamente utilizada em estudos sobre alterações climáticas com potenciais aplicações na monitorização da qualidade da água, denominada Ocean Color. O nosso objectivo é obter um modelo capaz de estimar com precisão a concentração de compostos activos da água do mar, a partir de medições de detecção remota da reflectância da superfície marítima, tendo em consideração o impacto da incerteza nas observações do sensor e nas aproximações do modelo.

Palavras-chave: Problemas de inversão não lineares, Representação e raciocínio da incerteza, Ocean color medições remotas, Aprendizagem profunda, Redes neuronais

Contents

List of Figures

List of Tables

Acronyms

ANN Artificial Neural Networks 6, 29

CNN Convolution Neural Networks 11, 27, 28, 29

GTN Generative Teaching Networks 20, 71

Huber Huber Loss 15, 41

INN Invertible Neural Networks xii, 27, 29, 30, 31
IOP Inherent Optical Propertie 32
IQR Interquartile Range 67

KDE Kernel Density Estimation xiii, 44, 45

MAE Mean Absolute Error 15, 41
MLP Multilayer Perceptron 6, 11, 31, 38, 39
MSE Mean Squared Error 15, 16, 41, 42, 51, 59, 70

OC Ocean Color 3, 4, 27, 32, 33, 48, 50, 64, 71

ReLU Rectified Linear Unit 11, 28, 39

SMOTE Synthetic Minority Over-Sampling Technique xii, 19

1

Introduction

In this chapter, it is made a brief introduction to what this document aims to achieve. It is divided in: Context (1.1), Objective (1.2), Application (1.3), Expected Contributions (1.4) and the Structure of the document (1.5).

1.1 Context

In the last century, scientists have extensively attempted to model the world. There are some mathematical models that through a set of parameters can foresee the behavior of a system. To predict the result of a measurement it is required a model of the system under investigation and a physical theory linking the parameters of the model to the parameters being measured. This is referred to as a direct problem. Direct problems in natural sciences are usually well-posed and well-conditioned. A problem is well-posed when: there is a solution, the solution is unique, and the solution depends continuously on the data [1] and is well-conditioned if small changes in the input lead to small changes in the output.

Direct problems usually have a relatively fast and easy computable solution, however, they have little to no real-life application, since they require the measurement of parameters that are exponentially more complex and expensive to attain than the observations they are able to model. Realistic problems involve working out unknown parameters and causes from observations of a system of interest, rather than modeling a system from known parameters, therefore a new research area emerged.

Inverse Problems consists in using the results of actual observations to infer the values of the parameters characterizing the system under investigation. Many problems of practical interest can be formulated as nonlinear inverse problems [2]. In a typical inverse problem, there is a set of model parameters, a set of observable parameters and a relation predicting the outcome of the possible observations. The model is typically a forward mapping f from the model parameters to the observable parameters. It allows predicting the results of measurements based on the model parameters.

If f is composed of linear equations, the inverse problem is linear. Otherwise, that is

most often, the inverse problem is nonlinear. Non-linearity and uncertainty play a major role in modeling the behavior of most real systems. When dealing with these problems a new complexity arises, as, unlike direct problems, inverse problems often are ill-posed as there may be no exact solution or multiple solutions, and ill-conditioned, small changes in the input lead may result in huge changes in the output, which paired with measurement errors on the observable data or the approximations made in the model specification result in great uncertainty and inaccuracy.

1.2 Objective

With this thesis, we aim to study the application of machine learning techniques to support reasoning for nonlinear inverse problems.

Through the employment of a forward model, it will be generated a large dataset that can be used to train a regression model that is able to approximate the inverse mapping of a given function. Artificial Neural Networks will be implemented as regression models, as they are capable of approximating continuous bounded functions and can be trained using the data obtained by the forward mapping. The research must also account for the inherent errors made in those approximations, as well as their influence on the underlying reasoning's uncertainty.

Initially simple problems will be addressed in order to study the efficiency of this methodology in this application and then applied to a more realistic problem explained in Section 1.3. The resulting work will then be compared to previously developed methodologies presented in Chapter 3 applied to the same problem.

To solve this problem this dissertation will address the following questions:

- How to map from the observable parameters to the model parameters using neural networks?
- How to generate data through the employment of the forward model to train the network?
- How can uncertainty of errors on the observable data be represented and reproduced to train the network?
- What is the efficiency of this method compared to previously developed ones?

1.3 Application

We aim to apply this thesis study to the Ocean Color (OC) investigation, a research field that studies the color of the ocean's water.

The oceans cover over 70% of the earth's surface and the life inhabiting the oceans plays an important role in shaping the earth's climate. Phytoplankton, the microscopic organisms on the ocean's surface, are responsible for half of the photosynthesis on the planet. These organisms at the base of the food web take up light and carbon dioxide and fix carbon into biological structures releasing oxygen.

Estimating the amount of microscopic phytoplankton and their associated primary productivity over the vast expanses of the ocean is extremely challenging to do from ships. However, as phytoplankton take up light for photosynthesis, they change the color of the surface ocean from blue to green.

Such shifts in ocean color can be measured from sensors placed high above the sea on satellites or aircraft, this methodology is called "OC remote sensing". In open ocean waters, the ocean color is predominantly driven by the phytoplankton concentration, and ocean color remote sensing has been used to estimate the amount of chlorophyll-a, the primary light-absorbing pigment in all phytoplankton [3].

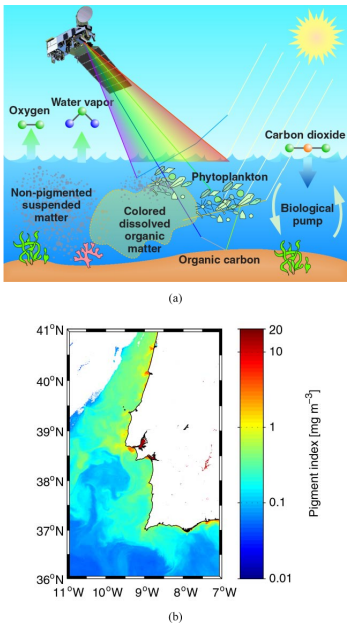


Figure 1.1: OC satellite and seawater compounds sensing

OC satellite missions can provide cost-effective environmental indicators at large spatial scales by deriving optically active seawater compounds (OC products) through remote sensing measurements of the sea-surface reflectance (Figure 1.1) [4].

A forward relation between the remote sensing reflectance (Rrs) at a given wavelength (λ) has been modeled in [5] and in order to estimate the amount of OC products, it is required to get its inverse function which is a real-world example of a complex nonlinear inverse problem and therefore one of the main objectives that this dissertation aims to solve.

1.4 Expected Contributions

With this dissertation we aim to provide a framework that can be used by members of the scientific community, allowing them to better understand how different approaches may behave when trying to support reasoning for nonlinear inverse problems. From the resulting work produced by this dissertation, we are expecting to provide:

- A set of recommendations concerning several different combinations of regression methods applied to the domain of nonlinear inverse functions;
- A library that is able to map the inverse of a given function through the employment of a neural network;
- A validation study regarding different approaches for nonlinear inverse problems;
- A model that is capable of accurately estimating the concentration of OC products, taking into consideration the impact of uncertainty on the sensor observations and the model approximations;

1.5 Structure

To better understand the proposed work, this report is divided into 3 chapters:

- Chapter 1, provides a contextualization of the problem at hand, explaining the motivation behind the need to solve it. A brief overview of the proposed solution is also presented here as well as the objectives behind it;
- Chapter 2 presents a brief introduction to basic concepts of Deep Learning and Data Sampling that will be of use in this thesis;
- Chapter 3 extends on the definition of Inverse Problems and introduces the current state of the art in solving them. This section also describes in more detail the intricacies of the main problem at hand and how we expect to validate our approach;
- Chapter 4 highlights the technical approach used to solve the problem presented in this dissertation;
- Chapter 5 presents the obtained experimental results during the development of this work;

2

Deep Learning

In order to better understand the work and research done, this chapter presents background knowledge about some fundamental concepts on deep learning and data sampling that will be used throughout this dissertation. Section 2.1 presents basic information on how deep learning algorithms work in order to establish a relation on the data. In Section 2.2 some Data Sampling techniques are presented as an adaptation will be applied in this thesis.

If the reader is familiar with these topics be free to skip to the next chapter, however, if any of these topics are not mastered, their reading is advised.

2.1 Fundamental Knowledge

Machine learning is a branch of artificial intelligence that allows computers to learn and improve on their own without having to be explicitly programmed. Machine learning is focused on the creation of computer programs that can access data and learn on their own [6]. Applying this to the inverse problems in study, we aim to find a nonlinear mapping T_Θ that is capable of defining a relationship from the observed parameters O to the model parameters M or an acceptable approximation of it.

$$T_\Theta(O) \approx M$$

The learning revolves on choosing an optimal combination of weights, Θ, given the training data, where the concept of optimality is quantified through a loss function that measures the quality of T_Θ [7].

As the main goal of this thesis is to study the application of deep learning techniques to support reasoning for nonlinear inverse problems and their efficiency, it is presented a series of concepts that will be crucial in order to comprehend this thesis.

Deep learning algorithms are a subset of machine learning, that consist of a structure of multiple layers, that try to learn representations of the data through a hierarchical composition of relatively simple nonlinear modules that transform features into progressively higher levels of abstraction.

Artificial Neural Networks (ANN) are neural computing systems inspired by the biological neural networks that constitute animal brains as both networks are composed of computational devices that are highly interconnected and this is what determines their functionality. An artificial neuron is the basis of a ANN. Each neuron receives a set of input signals ($X = x_1, x_2, ..., x_n$) and each input is associated with an adaptive weight ($W = w_1, w_2, ..., w_n$) that affects the impact of that input. The network is then trained to minimize errors between the desired value and the model's computed values through the adjustment of the weights.

2.1.1 Multilayer Perceptron

A Multilayer Perceptron (MLP) is a class of feed-forward Artificial Neural Networks. This architecture is one of the most basic and therefore most useful, making it suitable for the majority of all problems. It is composed of sets of neurons denominated by layers where each neuron connects to all neurons of the following layer with a certain weight.

MLP is a fully connected, multilayer feed-forward neural network. This is, each neuron of one layer receives as input the output of all neurons of the previous layer [8]. An A typical MLP is composed of three types of layers: an input layer that takes input from your dataset, an output layer that is responsible for outputting a value or vector of values that correspond to the format required for the problem, and an arbitrary number of hidden layers placed in between the last two that are the real computation engine of the network. The number of neurons in the input layer depends on the number of independent variables in the model, whereas the number of neurons in the output layer is equal to the number of dependent variables [9].

Even though the input and output layers are quite trivial and strict to define, the "middle" layers are a lot more complex and there is no correct answer. These layers are responsible for the computation of the network as they perform nonlinear transformations on the inputs received in order to compute the correct value. There are proofs [10] that a single hidden layer is capable of universal approximation. The universal approximation theorem states that a feed-forward network (like MLP), with a single hidden layer, containing a finite number of neurons, can approximate any continuous function, however, it does not specify how easy it is for that neural network to actually learn something and the computation time required, therefore additional layers may be helpful.

Having the number of hidden neuron layers is only part of the problem, the amount of neurons in each of these hidden layers must also be decided.

Using too few neurons in the hidden layers will result the model being unable to establish the relationship between the input and output variables accurately, generating a high error rate on both the training set and unseen data. On the other side, using to many neurons may induce multiple problems. One of them being overfitting, this is when the model simply over adapts to details of the training set that do not generalize to the universe from which the data was sampled. Another problem is the computation time it

takes to train the network, as this value can increase to the point that it is impossible to adequately train the neural network.

The network is then trained to minimize the error between the target and predicted values computed by the model. To improve accuracy the network adjusts the weights values, through a learning algorithm that involves two main steps, a forward-propagation step followed by a backward-propagation step.

The forward-propagation phase starts by attributing an input pattern to the input layer. The net input of a neuron j is the sum of each output of the previous layer, with size M, multiplied by its weights.

$$NET^j = \sum_{i=1}^{M} w_i * x_i$$

This value is then applied to a function denominated of activation function (Figure 2.1) and the result is the output of neuron j .

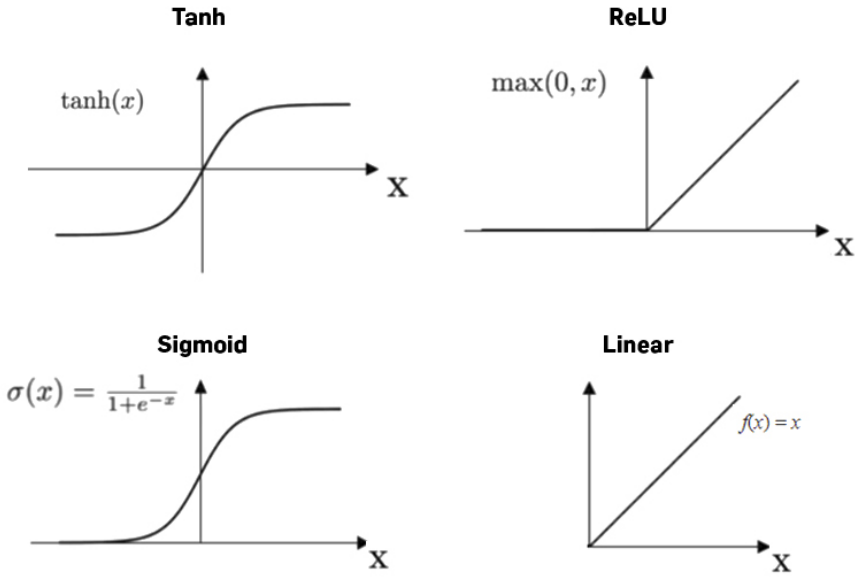


Figure 2.1: Activation functions and their general formula

Activation functions are a critical part of the design of a neural network. The chosen function in the hidden layer will control how well the network model learns the training dataset and the output activation function defines the type of predictions the model can make.

Typically the same activation function is used on all hidden layers, whilst the output layer may use a different activation function, as it depends upon the type of prediction required by the model.

The forward-propagation phase continues as activation level calculations propagate forward into the output layer through each hidden layer. In each successive layer, every neuron sums its inputs affected by the weights and applies onto the activation function to

compute its output. The output layer of the network then produces the estimated target value [9].

The learning algorithm then attempts to modify the weights W to minimize the error. To achieve this, the neuron is back-propagated for the appropriate weight adjustment. To know how to update the neuron's weights it is required to compute the derivative of the error as a function of the weights of the neuron.

Assuming it is used the common error function, sum of the squared errors (E) which for each neuron is calculated in regard to its target value t_k as:

$$E = \frac{1}{2}\sum_k (t_k - o_k)^2$$

and the used activation function is the sigmoid function :

$$f(NET) = \frac{1}{1 + exp(-NET)}$$

Then, calculating its derivative as a function of the weights, can be done using the chain rule for the derivative of compositions of functions:

$$\frac{\delta E_k}{\delta w} = \frac{\delta E_k}{\delta o_k}\frac{\delta o_k}{\delta NET_k}\frac{\delta NET_k}{\delta w}$$

$$\frac{\delta NET_k}{\delta w} = x_k \qquad \frac{\delta o_k}{\delta NET_k} = o_k(1 - o_k) \qquad \frac{\delta E_k}{\delta o_k} = -(t_k - o_k)$$

which is results in:

$$\frac{\delta E_k}{\delta w} = -(t_k - o_k)o_k(1 - o_k)x_k$$

In order to decrease the error, we have a learning rate $\eta \in [0, 1]$, that descends a small step in the error surface ($-\eta\frac{\delta E_k}{\delta w}$) used to update the new weight:

$$w_k^i = w_k^i + \eta(t_k - o_k)o_k(1 - o_k)x_k^i \tag{2.1}$$

The challenge is to determine a η that produces a satisfactory learning rate without jeopardizing the learning performance, as a large value for η can lead to instability in the network training and unsatisfactory learning, whereas an overly small value can lead to excessively slow learning.

One solution might be to initiate η with an high value and then decrease it as the learning session progresses.

2.1.2 Activation Function

There are many different types of activation functions used in neural networks, however, realistically only a small number of functions used in practice for hidden and output layers, being the most common Linear, Sigmoid (Logistic), Tanh (Hyperbolic Tangent) and ReLU.

2.1.2.1 Linear

The **linear activation function** is also called no activation. This is because the linear activation function does not change the weighted sum of the input in any way and instead returns the value directly.

$$f(x) = x$$

This is not a valuable function to be used on the hidden layer, as due to its linearity is unable to perform non-linear transformations to the data. However, this activation function is commonly used in the output layer for regression problems, as it is a unbounded function.

2.1.2.2 Sigmoid

The **sigmoid activation function** also known as the logistic function takes any real value as input and outputs values in the range [0,1].

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

This activation can be used in hidden layers, as it is able to perform valuable non-linear transformations to the input, or in the output layer, commonly used in classification problems as a form of displaying probabilities or in regression problems if a bounded domain is in study.

2.1.2.3 Tanh

The **hyperbolic tangent activation function** is also referred to simply as the Tanh function. This function is similar to the Sigmoid activation function, however the function takes any real value as input and outputs values in the range [-1,1].

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

The applications of this function is the same as Sigmoid.

2.1.2.4 ReLU

The **rectified linear activation function**, or ReLU activation function, is the most common function used for hidden layers. A major problem of the last two functions is that they saturate, this is, for extreme values they either snap to 1, if the input is positive, or to 0/-1, depending on the function, if the input is negative. In addition, the functions are only really sensitive to changes around their mid-point of their input. The limited sensitivity and saturation of the function happens regardless of whether the summed activation provided as input contains useful information or not. Once saturated, it becomes challenging for the learning algorithm to continue to adapt the weights to improve the model's performance [11].

The solution found was ReLU. This function simply returns the value provided as input directly, or the value 0 if the input is smaller or equal to 0:

$$ReLU(x) = max(0, x)$$

This activation function in considerably more desirable when training a neural network because its less computational extensive and it is not as unstable as the nonlinear ones, making it the ideal activation function to be used on hidden layers.

A small downside of ReLU is a problem called "dead neurons". A ReLU neuron is "dead" if it gets stuck in the negative side and always outputs 0. Because the slope of ReLU in the negative range is also 0, once a neuron gets negative, it is unlikely for it to recover. Which means that these neurons are not contributing in discriminating the input and are essentially useless. To combat this problem, the alternative found was variations of the ReLu function such as Parametric ReLU (PReLU or Leaky ReLu) or Exponential Linear (ELU) where for x > 0 the function is the same, however for x < 0 $PReLU(x) = \alpha x \quad ELU(x) = \alpha(e^x - 1)$.

2.1.3 Weight Initialization

As previously explained, each neuron in the network is composed of parameters referred to as weights used to calculate a weighted sum of the inputs. This weight are updated using the gradient descent algorithm (2.1), that continuously change the network's weights in order to minimize a loss function, resulting in a model that is capable of making accurate predictions. Weight initialization is the process that sets the initial value of weights of a model to small random values that define the starting point for the optimization. Using the right heuristic is able to reduce the training time and increase the overall network accuracy. There is no clear strategy when choosing this initial value, however it is obvious that there cannot be identical weights through all neurons, as this would result in equal parameter gradients and optimization across all neurons, it is necessary to break this symmetry from the beginning. Recurrent networks are more prone to instability if the initial weights are too large, and large initial weights may saturate activation functions or lead to other numerical issues. On the other hand, larger weights "spread out" the network more widely right away and are better at breaking neuronal symmetry. One common practice is to choose initial weights at random from a Gaussian distribution with mean zero and variance one. Other initialization schemes include taking into account the number of neurons in each layer, or the number of inputs and/or outputs, amongst other factors.

2.1.3.1 Glorot

Being i the number of inputs of a given neuron, **Glorot initialization method** computes its initial weight as a random number with a uniform probability distribution

between the range $[-\frac{1}{\sqrt{i}}, \frac{1}{\sqrt{i}}]$. These bounds become wider when the neuron has fewer inputs and more narrow with more inputs.

2.1.3.2 Normalized Glorot

Similarly to the aforementioned Glorot Initialization, the **Normalized Glorot Initialization method** is calculated as a random number with a uniform probability distribution, however, the range of the distribution not only takes into account the number of inputs i of a neuron, but also the number of outputs o of the layer. The normalized Glorot initialization calculates the weight as a random number with a uniform probability distribution between the range $[-\frac{\sqrt{6}}{\sqrt{i+o}}, \frac{\sqrt{6}}{\sqrt{i+o}}]$.

2.1.3.3 He

When used to initialize networks using the Rectified Linear Unit (ReLU) activation function, the Glorot weight initialization was found to have issues. Due to the popularity of this activation in the hidden layers of most MLP and Convolution Neural Networks (CNN) models, a modified version of the approach was created specifically for nodes and layers that use it. **He initialization method** is the current accepted method for initializing the weights of neural network layers and nodes that make use of the ReLU activation function. This heuristic computes the initial weight of a neuron as a random number with a Gaussian probability distribution where its mean 0 and has a standard deviation of $\sqrt{2/i}$, with i being the number of inputs of the neuron.

2.1.4 Optimizer

An optimizer is a technique or algorithm to adjust the various parameters used to minimize an loss function or to maximize the efficiency of production. Optimizers are mathematical functions that provide guidance on how to modify the neural network's weights and learning rate to minimize losses.

2.1.4.1 Gradient Descent

The most basic form of an optimization algorithm is the aforementioned Gradient Descent (sec 2.1.1). This algorithm adjusts each weight iteratively in order to minimize a given function to its local minimum.

$$w_t = w_{t-1} - \eta \frac{\delta E}{\delta w_{t-1}}$$

Gradient Descent iteratively moves towards the minima of the loss function by computing its derivative. This method requires the use of the entire training set to calculate the gradient of the cost function to the parameters which necessitates large amount of memory and slows down the process.

Stochastic Gradient Descent (SGD) solved the Gradient Descent problem by using only single records to updates parameters. But, still, SGD is slow to converge because it needs forward and backward propagation for every record. And the path to reach global minima becomes very noisy.

In order to combat this issues **Mini-Batch Gradient Descent** arose. This algorithm splits the training dataset into small batches and performs an update for each of those batches.

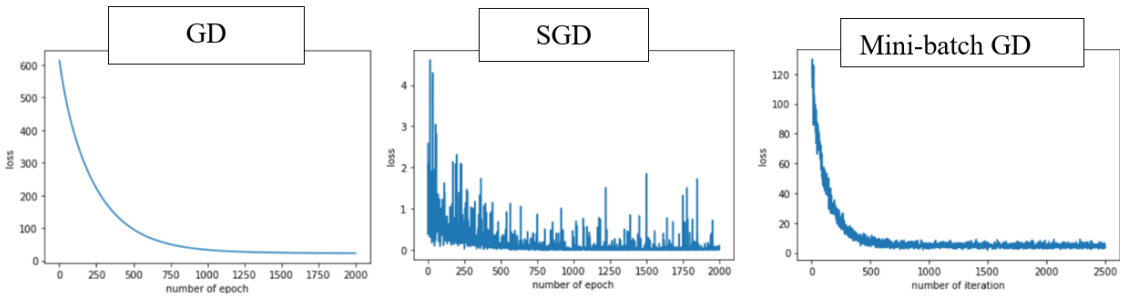


Figure 2.2: Comparison of loss decay between optimizers: Gradient Descent, Stochastic Gradient Descent and Mini-Batch Gradient Descent [12]

However these algorithms tend to be slow and unreliable as they get stuck in local minima, saddle points, or plateau regions. The following methods perform additional processing to make them faster and more reliable.

2.1.4.2 Momentum

The gradient descent with momentum algorithm simulates a moving object. Similarly to the inertia of a moving object, momentum also accumulates the gradient of the past steps to determine the direction to go while the current update gradient is used to fine-tune the final update direction.

$$w_t = w_{t-1} - \eta \frac{\delta E}{\delta w_{t-1}} + \beta * \Delta w_{t-1}$$

Typically the decay rate (β) is chosen around 0.9 as this value both permits the previous gradients to have an impact on the current update while also allowing to eventually stop.

This methodology is an optimization over the gradient descent algorithm as Momentum simply moves faster, because of all the momentum it accumulates and has the possibility of escaping local minimas as the momentum may propel it out of a local minimum.

2.1.4.3 RMSProp - Root Mean Square Propogation

A limitation of gradient descent is that it uses the same learning rate (η) for every input variable. **Adaptive Gradient Descent (AdaGrad)**, is an extension of the gradient descent

optimization algorithm that automatically adjusts this step size in each dimension based on the gradients seen for the variable (partial derivatives) over the course of the search. The reason behind this necessity is that the learning rate for sparse features parameters needs to be higher compare to the dense features parameter because the frequency of occurrence is lower.

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{\sum_{i=1}^{t}(\frac{\delta E}{\delta w_{t-1}})^2 + \epsilon}} * \frac{\delta E}{\delta w_{t-1}}$$

A limitation of AdaGrad is that it can result in a very small step size for each parameter, this is because the sum of gradient squared only grows and never shrinks, which by the end of the search that can slow down its progress so much it may not find the global optima. **RMSProp** fixes this issue by adding a decay factor.

$$v_t = \gamma * v_{t-1} + (1-\gamma) * \left(\frac{\delta E}{\delta w_t}\right)^2$$

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{v_{t-1} + \epsilon}} * \frac{\delta E}{\delta w_{t-1}}$$

The epsilon in both equations, is to ensure that we do not end up dividing by zero, and is generally chosen to be 10^{-10}. The decay rate (γ) indicates that recent gradients have a greater impact on the update, while older ones are essentially neglected.

2.1.4.4 ADAM - Adaptive Moment Estimation

ADAM optimizer is a combination of RMSProp and Momentum algorithms and its the most common go to when dealing with deep learning problems therefore the one we expect to give the best results in this thesis (Figure 2.3).

ADAM has two main parameters β_1 and β_2 which are the decay rate for the first and second moment respectively. The first moment, sum of gradients aims to simulate the speed of momentum:

$$v_t = \beta_1 * v_{t-1} - (1-\beta_1) * \frac{\delta E}{\delta w_t}$$

While the second moment, sum of gradient squared allows the ability to adapt gradients in different directions:

$$s_t = \beta_2 * s_{t-1} - (1-\beta_2) * \left(\frac{\delta E}{\delta w_t}\right)^2$$

To decide our learning step, we multiply our learning rate by average of the gradient and divide it by the root mean square of the exponential average of square of gradients:

$$w_t = w_{t-1} - \eta \frac{v_{t-1}}{\sqrt{s_{t-1} + \epsilon}} * \frac{\delta E}{\delta w_{t-1}}$$

The hyperparameter β_1 is generally kept around 0.9 while β_2 is kept at 0.99.

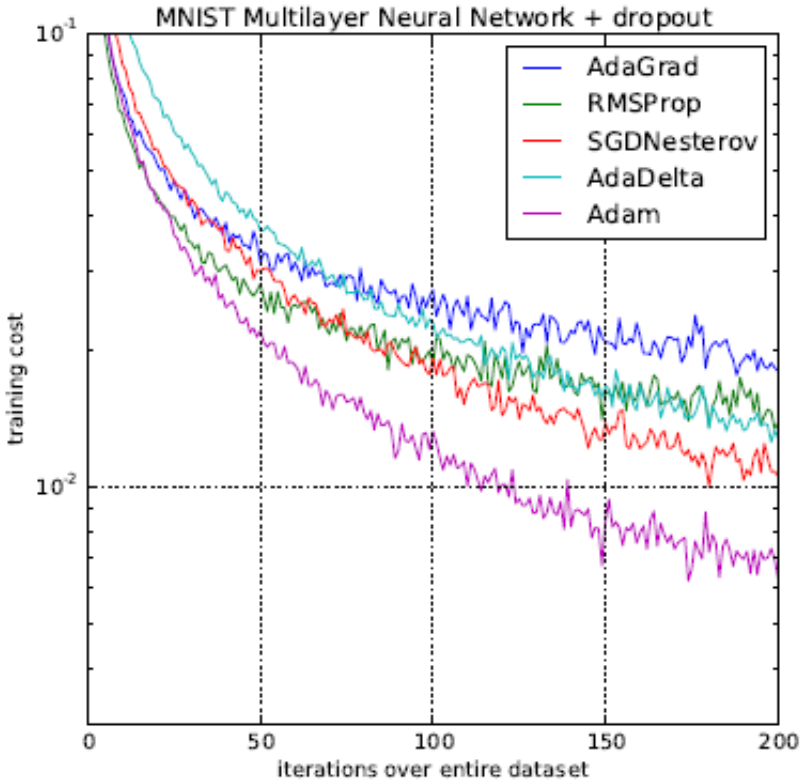


Figure 2.3: Optimizers training cost [13]

2.1.5 Loss Functions

There are two main problems that can be solved with deep learning, **Classification problems**, where we aim to predict in which category, from a discrete set, a given example belongs to. Or **Regression problem**, where this thesis falls in, which is the attempt to predict some continuous value given a certain input.

If we do this from a set of data containing the right answers, so we can then extrapolate to new examples, we are doing Supervised Learning. There are other other types of problems that can be solved with machine learning, such as clustering, for example, which is an example of Unsupervised Learning. While Supervised Learning requires that all data be labelled, Unsupervised Learning uses unlabelled data. Given our approach (a more thorough explanation ahead), we are able to acquire a fully labeled dataset, this is, every value of this dataset is perfectly labeled with the expected value the network should predict. Due to this, we will resort to training the network with the supervised learning approach since it has larger learning improvements compared to the unsupervised learning approach [7, 8].

Supervised-learning consists in estimating the values of Θ, through a training dataset composed of $(Y \times X)$-valued variables where X are the features related to some label Y. This can be formulated, as presented in [14], as minimizing a loss function $L(\Theta)$ which

can be represented by

$$L(\Theta) := F_{\mu}[d(T_{\Theta}(o), m)]$$

where d is a dissimilarity function that quantifies the quality of the values provided by the model T.

Depending on the task at hand, d can take various forms, however for Regression problems, Mean Squared Error (MSE), Mean Absolute Error (MAE) and Huber Loss (Huber) are the most commonly used.

2.1.5.1 Mean Squared Error

When we are training a model, we aim to learn a probability distribution P_{model} that can best describe the actual distribution of the data P_{data}. So, we want to find the set of weights θ that maximize the obtained probability when we give the model the training set X. Thus, given our dataset $X = \{x^t, y^t\}_{t=1}^{N}$ and knowing that $p(x, y) = p(y|x)p(x)$ the likelihood of θ is:

$$\ell(\theta|X) = \prod_{t=1}^{n} p(x^t, y^t; \theta) = \prod_{t=1}^{n} p(y^t|x^t; \theta) \times \prod_{t=1}^{n} p(x^t; \theta)$$

In order to choose the best hypothesis we pick the one with the maximum likelihood.

Due to the possibility of numerical issues, its actually easier to maximize the logarithm of the formula above. And as $p(x; \theta)$, the probability of drawing those x values in our data from the universe of possible values, is constant across all hypotheses it can be discarded.

$$L(\theta|X) \propto log \prod_{t=1}^{n} p(y^t|x^t; \theta)$$

Assuming that the probability of obtaining some y value given some x is approximately normally distributed around our prediction

$$p(y|x) \sim N(P_{model}(x, \theta), \sigma^2)$$

We have that

$$L(\theta|X) \propto log \prod_{t=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{[y^t - P_{model}(x,\theta)]^2}{2\sigma^2}}$$

which can be simplified as:

$$L(\theta|X) = -\sum_{t=1}^{n} [y^t - P_{model}(x, \theta)]^2$$

Thus, this implies, under our assumptions, that to find the hypothesis with the maximum likelihood we need to find the hypothesis with the minimum squared error on our training set. For this reason, a common choice for the dissimilarity function d, is to use the Mean Squared Error function:

$$MSE(\Theta) = \frac{1}{n} \sum_{t=1}^{n} (y^t - P_{model}(x, \theta))^2$$

The MSE function is very sensitive to outliers, as the error grows quadratically in this function, which means that outliers in our data will contribute to much higher total error. However this function is "smooth" as it is differentiable in its whole domain making it easy to optimize.

2.1.5.2 Mean Absolute Error

The Mean Absolute Error function computes the amount of error in the predictions. It is the difference between the predicted value and true value.

$$MAE(\Theta) = \frac{1}{N}\sum_{i=1}^{N}|m_i - \hat{m}_i|$$

This loss is more robust to the effect outliers as each residual contributes proportionally to the total amount of error, meaning that larger errors will contribute linearly to the overall error. However this function is not differentiable for $x = 0$, and this point of discontinuity makes it difficult to optimize.

2.1.5.3 Huber

The Huber Loss function is a combination of the squared error and absolute error functions. It is a mean absolute error, that becomes quadratic when the error is smaller than a controlled hyperparameter δ that can be tuned.

$$L_\delta = \begin{cases} \frac{1}{2}(m-\hat{m})^2 & if\ |(m-\hat{m})| < \delta \\ \delta((m-\hat{m}) - \frac{1}{2}\delta) & otherwise \end{cases}$$

The choice of the delta value is critical because it determines what it is willing to consider an outlier. Hence, the Huber loss function could be less sensitive to outliers than the MSE loss function, depending on the hyperparameter value.

This reduces the effect of outliers as if a point is considered an outlier (this is done by configuring the hyperparameter δ) its effects on the loss function are as the MAE. However if the point is not an outlier, we take advantage of the ease of computing the gradient of the MSE.

When done right, the tunning of this hyperparameter allows us to ignore measurements where the "noise" is too great as a faulty measurement and give it little to no importance on the update of the network.

2.1.6 Regularization

Having a method that is able to compute the hypothesis with the minimum error on our training set may not result in an optimal training, as this set does not represent the entire domain of possible data, therefore only improving the model in regard to the best fit the training set will eventually increase the test error and consequently decrease the model accuracy.

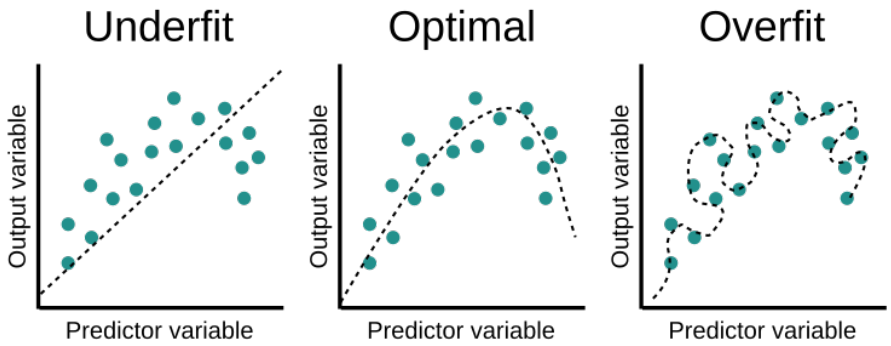


Figure 2.4: Training unbalances

This is usually referred as *overfitting* where the model simply over adapts to details of the training set that do not generalize to the universe from which the data was sampled.[7]. In order to mitigate this, in this future subsection we present some techniques that are able to mitigate this occurrence.

2.1.6.1 Early Stopping

A way of regularizing neural networks is to stop training in an earlier stage, to prevent overfitting from getting worse. As the optimizer moves the parameter vector away from the initial values (close to 0) it can improve the fit in the training set while degrading performance on the validation set and therefore worsening the predictions of the network. Early stopping keeps the optimization closer to the initial values, thus restricting this transition into more extreme values that worsen overfitting [8].

2.1.6.2 Tikhonov Regularization

Tikhonov regularization, also known in the machine learning community as Ridge regression, aims to mitigate the overfitting problem while assuring that the network does not loose performance.

This method revolves on penalizing the coefficients for those input variables that do not contribute much to the prediction task. In general, this method provides improved efficiency in parameter estimation problems in exchange for a tolerable amount of bias. This is achieved by minimizing the sum of two parameters, the error provided by the model and a penalty function on the parameter vector.

$$J(\Theta) = MSE(\Theta) + \alpha \frac{1}{2} \sum_{i=1}^{N} \theta_i^2$$

In this function α is the adjustable regularization parameter, who gives greater or lesser weight to the regularization. A high value of α results in a higher weight on the regularization and therefore less importance to minimizing the training error resulting on a flatter model, while a lower value results on less regularization with an higher emphasis

on minimizing the training error [8, 15]. The adjustment of the value of α will be a subject to study during the development of this thesis.

2.1.6.3 Dropout

Dropout is a way of simulating the training of different networks by creating sub-sets of working neurons. To accomplish this, during training, random neurons are "deactivated"by ignoring their weights and activation. Each neuron of the hidden or input layers has a probability of getting ignored, with a value of approximately 0.5 and 0.2 correspondingly. With this approach, the network is considerably different in each iteration as we have different active neurons during training. After training, the activation of neurons or weights in the case of forward neurons, can be re-scaled by the dropout probability to retain the same expected activation [8].

2.2 Sampling the Data

When dealing with data that is comprised of many features, we have to take into account that most of them might not be desirable due to them being uninformative. Even if none of the features existent in our data is irrelevant, by having too many features we might obtain a model that is especially suited to perform well over the training data but will perform poorly over data that is not similar to the training data impacting the performance of the network [7].

However in the context of this thesis, in order to have a dataset to train, test and validate the network, we propose to produce this data through the employment of the forward model that we are trying to invert.

When generating data it is important to create a balanced and representative dataset to train an accurate network. When this does not happen a problem emerges and when dealing with imbalanced datasets a model becomes biased towards the dominant domain.

In this thesis will be studied possible approaches to generate a training dataset that is sufficiently representative and balanced in order to properly train the network

2.2.1 Random Sampling

A naive approach to provide an initial dataset in order to train the network can be to randomly generate the data. This naive technique would have per basis the a priori information on the model parameters, therefore providing some feasible values, however, this approach would not work in the long run. As previously explained, this thesis will be analysing nonlinear inverse problems, therefore, when randomizing data without any kind of information, the most likely outcome would be an unbalanced dataset, with a nonuniform distribution, that would not be capable of accurately training the network. Another downside of this technique is the fact that, even though the data generated might be feasible as we are sampling from previously measured parameters ranges, the

likelihood of them happening can be very small, providing therefore an unrealistic set of data that affect the efficiency of the network when working with real-life measurements.

2.2.2 SMOTE

As previously explained, random data sampling can provide us a dataset without a uniform distribution, where too few examples of edge cases exist for a model to effectively learn the decision boundary.

This problem is quite common in classification problems, where a minority class is overshadowed in a dataset. One way to solve this problem is to oversample the examples in the minority class. This can be achieved by simply duplicating this class examples in the training dataset prior to fitting a model. This can balance the class distribution but does not provide any additional information to the model.

SMOTE, is a technique that attempts to solve the unbalance problem while also providing a more robust training data to the network. SMOTE algorithm selects a random example from the minority class. Then k of the nearest neighbors for that example are found. A randomly selected neighbor is chosen and a synthetic example is created at a randomly selected point between the two examples in feature space (Figure 2.5)[16].

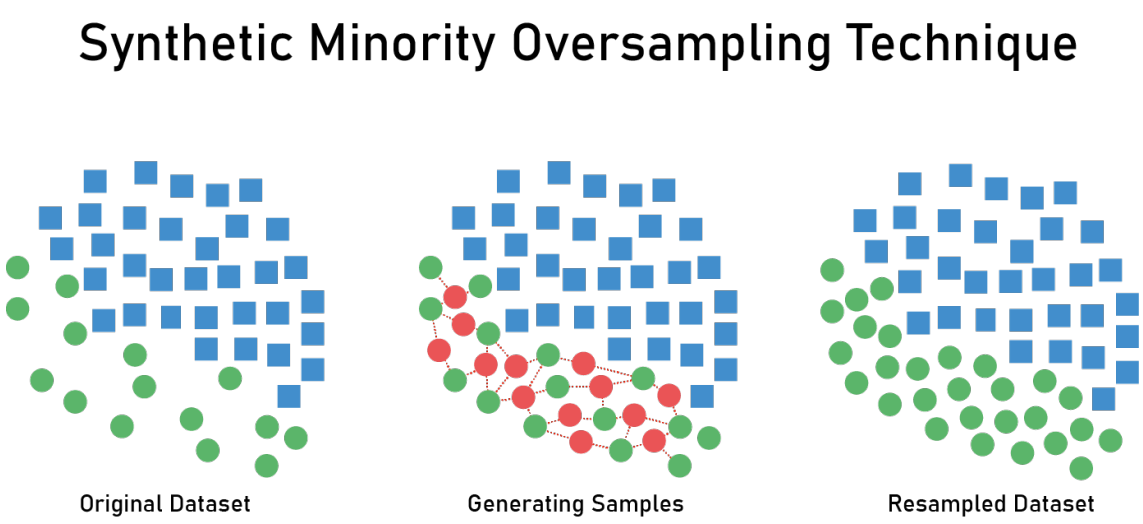


Figure 2.5: Basic principle of SMOTE

Since this in thesis we are working with a regression model and not a classification model, a direct application of SMOTE is not useful. The problem presented in our approach is not that of a presence of a minority class, but a dataset that is not representative of the entire domain. We believe that an adaptation of the SMOTE technique may be able to mitigate it. To values of the observed space that are not represented in our dataset, we would select the nearest points and through the analysis of their model parameters we would attempt to generate a set of points in the unrepresented domain. By applying this method we are able to produce a more balanced and uniform dataset that translates into a more efficient network training.

2.2.3 Generative Teaching Networks

One viable approach can be allowing the network to train itself. Generative Teaching Networks (GTN) is a neural networks architecture that generate data and training environments that are used later in the training phase. GTNs have two major phases **Generating data and training** and **Testing on real data**. In the first phase, the generator produces completely artificial data (that can not be a realistic dataset) that is used to train the neural network. Afterwards, the network is tested on real data where its gathered meta-loss information that is later used to improve the generator and improve the data. The network is then discarded and the process repeats (Figure 2.6) [17].

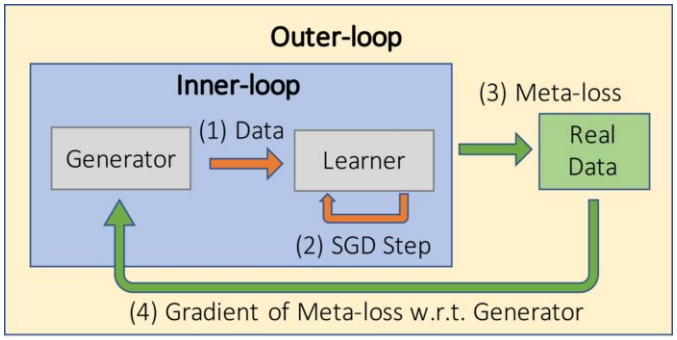


Figure 2.6: An overview of generative teaching networks. The generator (a deep neural network) generates synthetic data that a newly created learner neural network trains on. After training on generated data, the learner is able to perform well on the target task despite never having seen real data[17].

This approach is not totally applicable to this thesis problem as we neither need to train the generator nor have real data to provide a meta loss function with regard to the generator, however, a similar approach might be viable in our context. As we already possess the forward function it is not required to improve or update the generator, however, in this case we will have an adaptive sampling generator, this is, initially it will be generated a random dataset that shall be split in two. The first set, like in GTNs, will be used to train our network and the second one to test it, then, through the results of the test we will adapt and balance the generator to create new data in the domain that the network currently has a higher fail rate and repeat the process until a accurate network is formed.

3

Inverse Problems

In this chapter, it is presented some background knowledge on fundamental concepts of Inverse Problems (Section 3.1) and a critical analysis of previous methodologies present in the literature to solve them, both probability approaches (Section 3.2) and deep learning approaches (Section 3.3). Finally, in this chapter we better introduce the main problem we aim to solve, the complexities it possesses, and a validation method for our approach (Section 3.4).

3.1 Definition

Inverse problems appear when we want to see or examine something that we cannot access directly. What we have are indirect observations of a quantity of interest. Therefore the definition of an inverse problem is that of a mapping between objects of interest and acquired information about these objects (Figure 3.1).

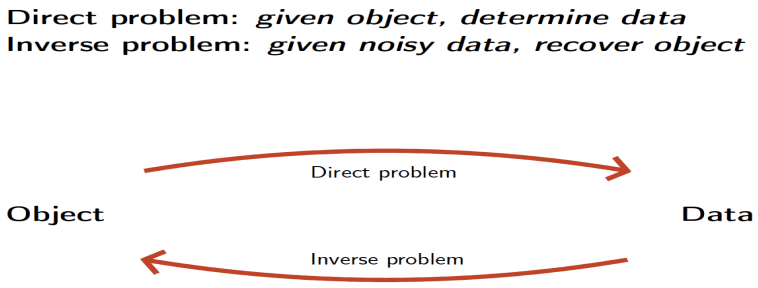


Figure 3.1: Informal definition of a Direct and Inverse Problem

In a typical inverse problem, there is a set of n model parameters $M = m^1, m^2, ..., m^n$, a set of k observable parameters $O = o^1, o^2, ..., o^k$, and a relation $o^i = f^i(m^1, m^2, ..., m^n) + \epsilon$ predicting the outcome of the possible observations where ϵ is the uncertainty related to measurement errors or approximations made in the model specification. The model is a forward mapping f from the model parameters to the observable parameters. It

allows predicting the results of measurements based on the model parameters. Solving the inverse problem amounts to finding points $m \in M$ from knowledge of the data $o \in O$ such that the relation $\boldsymbol{f}$ or an approximation of it holds.

3.1.1 Example

An example of a direct problem is the propagation of seismic waves produced by an earthquake. Given the epicenter coordinates (ex, ey) and the propagation speed of the wave (v), we can easily compute the arrival time of the seismic wave (ts) to a seismic station with coordinates (sx, sy), by calculating the distance between the two points and divide it by the speed of the wave:

$$ts = \frac{\sqrt{(ex - sx)^2 + (ey - sy)^2}}{v}$$

.

However, this problem has little to no real-life application, since its difficult to measure the epicenter of a seism and on the other hand its relatively easy to obtain the time that a wave hit a seismic station. Therefore a more pragmatic problem would be to through the time that the wave reached the seismic station compute the epicentral coordinates of the earthquake.

In this case, the observable parameters are the time that the wave reached the seismic station and the seismic station position whilst our model parameter is the epicentral coordinates of the earthquake which we aim to compute a function f that is able to predict.

A plausible model to estimate the epicenter would be:

$$(ex - sx)^2 + (ey - sy)^2 = (v * ts')^2$$

where $ts' = ts \pm \sigma$ and σ being the experimental uncertainties. The resulting feasible region of this nonlinear function would result in an annulus shape that has an infinite set of solutions, making it impossible to accurately pinpoint the epicentral coordinates of the earthquake.

In order to provide a more precise estimation of the epicenter, we can resort to having multiple seismic stations (S) and intersecting the resulting sets.

$$\bigcap_{s \in S} (ex - sx)^2 + (ey - sy)^2 = (v * ts')$$

Applying our model to an example provided by Albert Tarantola in [2] using constraint programming, where a seismic wave produced by the explosion has been recorded at a network of six seismic stations is presented in table 3.1 where the rows are their coordinates in a rectangular system at observed arrival times respectively:

It is assumed that seismic waves travel at a constant velocity of $v = 5km/s$ and experimental uncertainties on the arrival times are independent and can be modeled using a Gaussian probability density with a standard deviation $\sigma = 0.1s$

(sx,sy)	(3 km , 15 km)	(3 km , 16 km)	(4 km , 15 km)	(4 km , 16 km)	(5 km , 15 km)	(5 km , 16 km)
ts	3.12	3.26	2.98	3.12	2.84	2.98

Table 3.1: Positions of each seismic station and relative time the seism was sensed

In Figure 3.2 the gray area is a cover of the possible epicenter coordinates of the seismic event assuming that the observation errors cannot exceed 3σ and the red dot is the epicenter coordinates that minimize the sum of the squared errors.

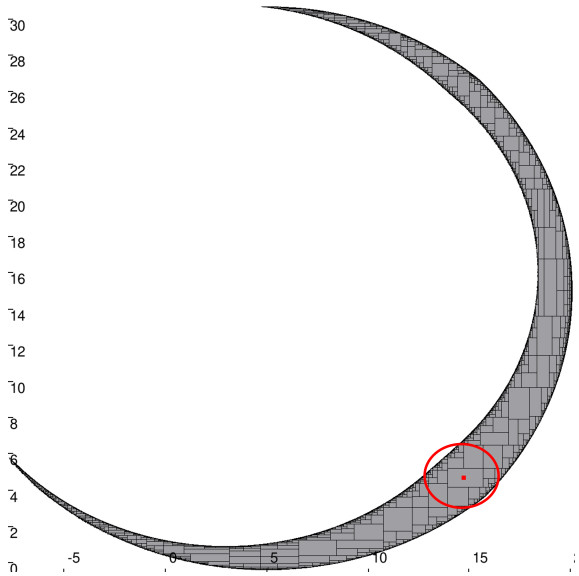


Figure 3.2: Epicenter calculation through constraint programming

3.1.2 Problem

As stated by Hadamard [18], a problem is well-posed if and only if it conforms to three conditions: existence, uniqueness, and stability.

- **Existence** there should be at least one solution.
- **Uniqueness** there is at most one solution.
- **Stability** the solution's behavior changes continuously on data.

If the forward mapping is bijective and allows a continuous inverse f^{-1} then the inversion satisfies all these conditions and therefore is a well-posed inverse problem. The condition of existence is broken if the measured noisy data does not belong to the domain of the forward map, $f(m) + \epsilon \notin O$. Uniqueness fails if two different quantities m_1 and m_2 return the same measurement, $f(m_1) = f(m_2)$. Stability is a condition in which a slight disturbance in a system does not produce too disrupting an effect on that system. In terms of the solution of a differential equation, a function $f(x)$ is said to be stable if any other solution of the equation that starts out sufficiently close to it when x = 0 remains close to it for succeeding values of x [15].

When the model equations are nonlinear, the problem is a nonlinear inverse problem. Usually, this family of problems are ill-posed problems as they may have no exact solutions (no combination of parameter values are capable of predicting exactly all the observed data), and solutions are not necessarily unique (different combinations of parameter values may induce the same observable values) and the stability of solutions is not guaranteed (a small change in the observed data may induce arbitrarily large changes in the model parameters). This ill-posedness status can be seen in the aforementioned example (Figure 3.2) as even though we can predict a possible epicenter we still have a large degree of uncertainty as we are only sure that it is in the vast gray area, therefore, we have multiple solutions and with a slight increase of the measurement errors, the area of the circle increases exponentially. For this thesis, we will restrict the study to nonlinear inverse problems that satisfy the existence and uniqueness conditions (are not ambiguous) but, due to nonlinearity, the inverse function f^{-1} cannot be computed analytically and the uncertainty on the measurements may induce a large uncertainty on the solutions.

3.2 Solving Inverse Problems using Probabilistic Approaches

Depending on the approaches for solving inverse problems, the solutions are either estimates, bounding values, or probability distributions of the model parameters. In this section we cover methods for these three approaches to inverse problems:

- nonlinear regression methods that provide an estimate of the model parameters;
- bounded error estimation that gives guaranteed bounding values for the model parameters;
- other stochastic approaches that provide an a posteriori probability distribution of the model parameters;

Most of the work covered in this section can be found in [19].

3.2.1 Best-fit Approaches

Nonlinear regression methods are used in traditional approaches to solve nonlinear inverse problems, these attempt to search for the model parameter values that best fit a given criterion. The regression considers a function that from a set of independent variables x and the model parameters m predicts the value of the dependent variable y.

So when having a set of size k of system observations (x_i, y_i) and assuming a measurement error ϵ_i on the observed y_i, the regression model is:

$$y_i = f(x_i, m) + \epsilon_i, \qquad 1 \leq i \leq k$$

Nonlinear regression methods look for model parameter values that minimize a suitable criterion based on several assumptions about the distribution of errors ϵ_i, where

appropriate analytic techniques can be used to characterize the uncertainty around the obtained parameter values with additional assumptions on the regression model. The least squares criterion, for example, minimizes a quadratic norm of the difference between the vector of observed data and the vector of model predictions:

$$\sum_{i=1}^{k}(y_i - f(x_i, m))^2 = \sum_{i=1}^{k}\epsilon_i^2, \qquad 1 \leq i \leq k$$

If errors ϵ_i are independent and normally distributed with zero mean and constant variance, then the least squares estimator is the maximum likelihood estimator, it estimates the values of the model parameters that produce a distribution that gives the observed data the greatest probability. Additionally, if the mapping f is linear with respect to the model parameters, then the values of the parameters given the observed data are necessarily normally distributed and confidence regions can be analytically computed.

Even though the least squares approach might be adequate for specifying the uncertainty on the model parameters given the observations when working with linear problems, this is not the case for more complex problems, like the nonlinear problems this thesis proposes to solve, as the minimization function may have multiple local and global minimums. Without an explicit formula for obtaining the best-fit values, minimization is usually performed through local search algorithms. If such algorithms converge to a local minimum, the solution obtained is no longer the maximum likelihood solution and even if one is found, the probability distribution of the model parameters given the observations may not be realistically approximated by a normal distribution and so the computation of any confidence regions based on such assumption will be inadequate.

3.2.2 Constraint Approaches

In contrast to the best-fit approaches that attempt to provide a single scenario which may be inaccurate for the characterization of the parameters, constraint programming provides a framework to characterize the set of all possible scenarios that are consistent with the constraints of a problem given the uncertainty on its parameters.

A classical constraint approach to solve inverse problems is known as bounded-error estimation [20]. The idea is to replace the search for a single best-fit solution with a characterization of the set of solutions that are consistent with predetermined acceptable measurement errors on the observed data.

Bounded-error estimation assumes a reliable bound for each measurement error ϵ_i, namely $a_i \leq \epsilon_i \leq b_i$, and applies constraint solving techniques to compute a the feasible space of:

$$y_i - b_i \leq f(x_i, m) \leq y_i - a_i, \qquad 1 \leq i \leq k$$

This approach, however, possesses two major downsides, the first one is that it considers all the consistent solutions of the same likelihood therefore it becomes unable to distinguish between them making it inadequate to characterize the uncertainty on the

parameter values given the observed measurements, the second one is that due to the nonlinearity of the function, the uncertainty of measurements may produce a large set of consistent solutions, deteriorating, even more, the previous downside.

3.2.3 Stochastic Approaches

Traditional Monte Carlo techniques can be used to deal with nonlinear inverse problems that are inadequate to be solved with best-fit approaches.

Such stochastic alternatives [2] perform a kind of Bayesian reasoning where the a priori information on the model parameters is transformed into an a posteriori probability distribution, by incorporating the forward model and the actual uncertainties. The solution is not a single scenario but a collection of scenarios consistent with the data and the a priori information. Extensive random sampling is used to obtain the collection of consistent scenarios necessary to characterize the a posteriori distribution of the parameter values. However, even after intensive computations, such characterization may be inaccurate, because a significant subset of the probabilistic space may have been missed. These approaches cannot prune the sampling space based on model information which can be a significant drawback in certain nonlinear problems.

3.2.4 Hybrid Approaches

In [19], Carvalho et al, propose a hybrid approach that combines the stochastic representation of uncertainty on the parameter values and a reliable constraint framework robust to nonlinearity. It associates an explicit probabilistic model with the problem, similar to stochastic approaches, and assumes reliable bounds for measurement errors, similar to constraint approaches. This approach computes conditional probability distributions of the model parameters, given the noisy observations and the forward model.

Once established a probabilistic model and a set of constraints over the model parameters, the probabilistic constraint approach relies on probabilistic constraint programming to compute any statistical information on the model parameters consistent with the experimental results. To enforce consistency with the model constraints, the probabilistic information is conditioned to its feasible space. The probability of the model parameter values is computed as their conditional probability given the feasible space.

The framework requires the computation of the integral of a distribution function over the region of interest (delimited by constraint propagation) and two options are offered: a certified method, using advanced quadrature techniques based on Taylor models [21]; an approximate method, using a Monte Carlo based technique [19].

Whereas the certificate method may be too computationally intensive and difficult to scale for larger parameter sizes, the approximate method may attain quite accurate results depending on the pruning of the sampling space achieved through constraint propagation.

This approach was then used to solve the OC problem this thesis also proposes to solve therefore, we will compare the results of our approach with the ones obtained with the probabilistic constraint approach available in [19].

3.3 Deep Learning Approaches

There have been previous attempts to tackle Inverse Problems using Deep Learning techniques. One of the most common architecture used these are the CNN, as in [14, 22–24]. This architecture is very useful to solve inverse problems that aim to find the model parameters when the observed parameters are contained in some high-dimension type of data, such as images (e.g. X-ray computed tomography), videos, sounds, and others. This is due to the fact that this architecture incorporates a pre-processing phase to retrieve features within the learning itself, which can result in features that can better capture the structure of the data and that ultimately will result in a better performing algorithm.

Another architecture that has been applied to ill-posed inverse problems by Ardizzone et al, in [25, 26], is the use of INN. This type of architecture focus on learning the forward process, using latent output variables to capture information that would have been lost when working with ambiguous inverse problems, where INN thrive. Due to this architectural nature, a model of the inverse process is learned implicitly.

In this section, it will be presented in more detail, how these architectures function as well as a critical evaluation on why this may or may not be a good solution to the problem this thesis proposes.

3.3.1 Convolution Neural Networks

Convolution Neural Networks (CNN) are deep neural network architectures that have achieved groundbreaking results related to pattern recognition by the successive application of convolution layers to its input. There are two main stages in a CNN model: the feature extraction/learning and the classification stage (Figure 3.3).

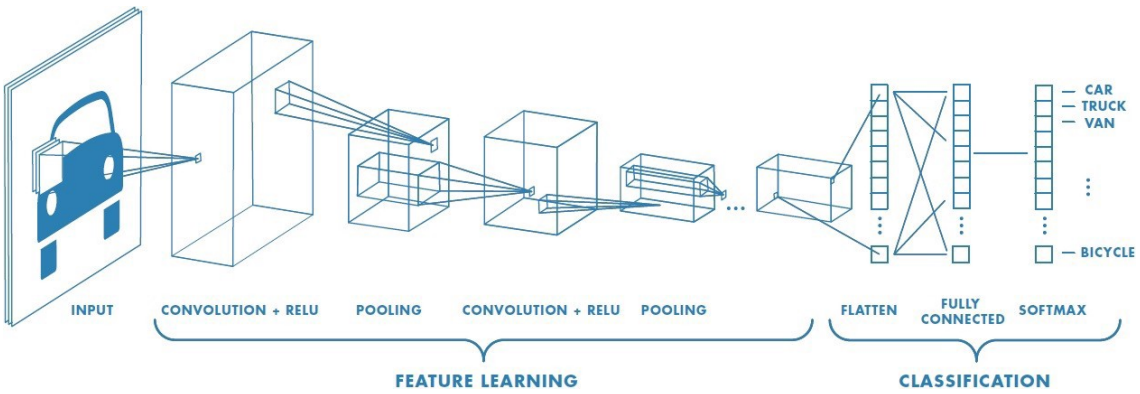


Figure 3.3: Schematic of an example of a CNN model [27, 28]

In the first stage, it is intended to reduce the input into a form that is easier to process while maintaining every important feature information. This is achieved by applying multiple successive convolution layers to the input like a filter [29]. This filter is composed of a kernel, a stride, padding, and an activation function. The kernel is a small matrix of weights, with a defined height and width smaller than the image to be convolved, that is applied to each section of the input that it covers, and performs a matrix multiplication between itself and this subsection of the input. The kernel will move according to a stride value, which corresponds to the number of pixels shifts over the input matrix.

Once the input is completely transversed, from these convolutions we obtain a single feature map. The feature map size depends on the used kernel size, stride, and padding. This process can be seen in Figure 3.4. An activation function is the last component of the convolutional layer to increase the nonlinearity in the output. Generally, ReLU function or hyperbolic tangent function are used as an activation function in a convolution layer.

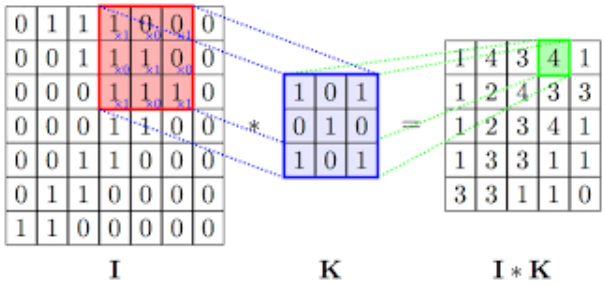


Figure 3.4: Convolution of 8x7 input (I) with 3x3 kernel (K) and a stride of one pixel

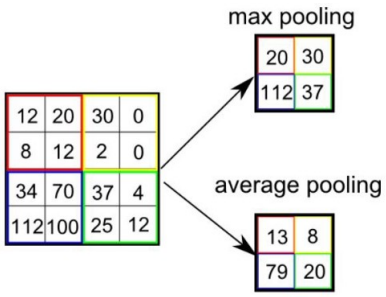


Figure 3.5: Types and Effect of Pooling

Another mechanism that CNNs use in order to reduce the size of the convoluted features, even more, is the usage of pooling layers. Pooling layers reduce a portion of the image covered by the Kernel to a single value. (Figure 3.5).

CNNs use multiple filters in parallel for a given input resulting in different specialized features for the same input, which will provide a more complete and diversified final collection of features to be used during the classification phase. Also, by subsequently using an extracted feature map from one convolution layer to be the input of the next, we allow a hierarchical decomposition of the input [29].

The second stage, also known as the classification stage, uses the flattened extracted features from the model, provided by the previous stage, and uses them to train a ANN.

The complexity that all [14, 22–24] have in common, is that their observed parameters are contained in multiple-dimensional data (i.e. images,video,sound,...), and therefore it was required an architecture that is able to extract valuable features that would be able to train their network. This is where CNN architecture thrives, the extraction and combination of low-level features, such as edges and color, to more complex higher-level which would then be used to perform their task. However, this is not applicable to the problem this thesis aims to solve due to the fact that feature extraction, from a dataset or image, is not a concern to our approach since we will be generating data through the employment of the forward model that only consists in features and measurements that we are already interested in. For this reason, CNNs will not be applied in this thesis.

3.3.2 Invertible Neural Networks

Invertible Neural Networks (INN) focus on learning the forward process, using additional latent output variables to capture the information that otherwise would be lost. This type of network allows it to be trained on the well-defined forward process, which is computationally easier than the inverse, and obtain the latest by running them backward at prediction time (Figure 3.6).

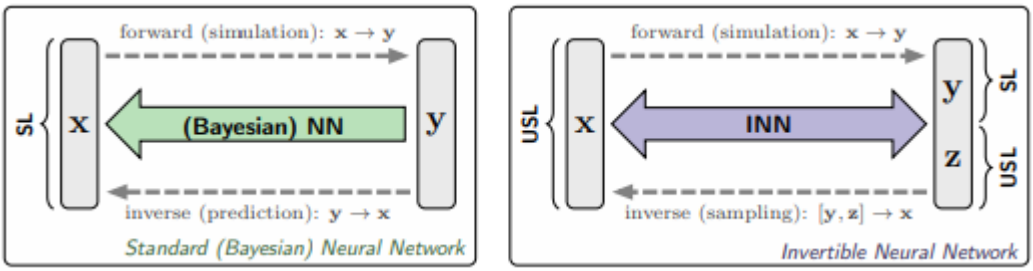


Figure 3.6: Comparison between a Standard Neural Network and an Invertible Neural Network [30]

The INN architecture presented in [30], has as the basic unit of this network a reversible block consisting of two complementary affine coupling layers. A block's input vector u is split into two, u_1 and u_2, which are transformed by learned functions s_i and t_i ($i \in 1, 2$) (Figure 3.7 top scheme), where $\odot$ is element-wise multiplication. The output is just the concatenation of the resulting parts of v_i. With some rearrangements, we can recover u_i from v_i to compute the inverse of the whole affine coupling layer (Figure 3.7 bottom scheme) where $\oslash$ is element-wise division.

Due to direct division numerical problems, in practice, it is used the exponential function, therefore, clipping the extreme values of s_i. The resulting forward process is as follows:

$$\mathbf{v}_1 = \mathbf{u}_1 \odot \exp\left(s_2(\mathbf{u}_2)\right) + t_2(\mathbf{u}_2)$$

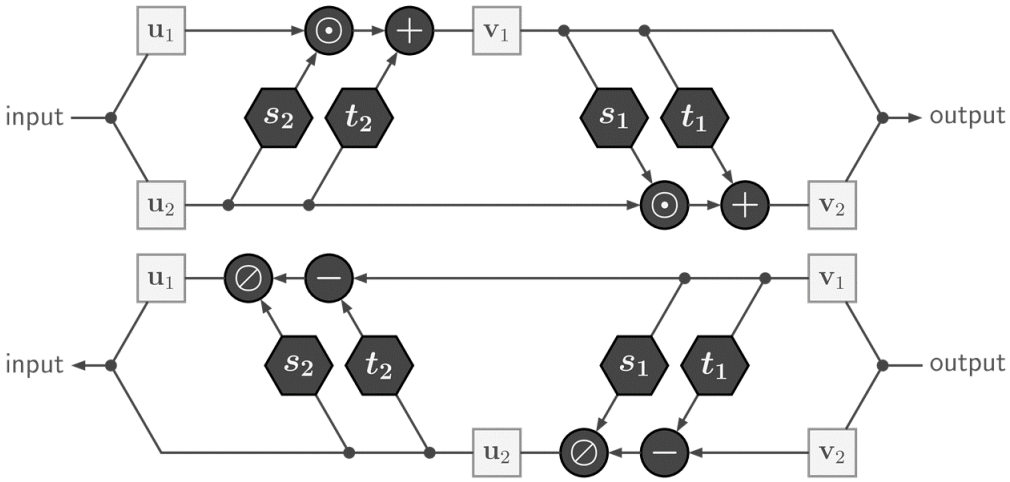


Figure 3.7: INNs forward (top) and inverse (bottom) processes

$$\mathbf{v}_2 = \mathbf{u}_2 \odot \exp\Big(s_1(\mathbf{v}_1)\Big) + t_1(\mathbf{v}_1),$$

And the inverse process:

$$\mathbf{u}_2 = (\mathbf{v}_2 - t_1(\mathbf{v}_1)) \odot \exp\Big(-s_1(\mathbf{v}_1)\Big)$$

$$\mathbf{u}_1 = (\mathbf{v}_1 - t_2(\mathbf{u}_2)) \odot \exp\Big(-s_2(\mathbf{u}_2)\Big)$$

In order to combat the imminent information loss present when computing the inverse process of ambiguous models (Figure 3.8- 1 and 2), INN introduces an additional latent output variable z, that captures the information of x that is not contained on y (Figure 3.8- 3). As opposed to Standard Neural Networks where x is associated only with y, INN's learn to associate hidden parameters of x with the unique pair [y,z] of measurements and latent variables. Forward training then optimizes the mapping $[y,z] = f(x,\theta)$ and implicitly determines its inverse $x = f^{-1}(y,z) = g(y,z,\theta)$.

One downside of this INN architecture is that in order to enforce the relation $f = g^{-1}$ both its input and output vectors of each module must have the exact same dimensionality. So, if the nominal dimensions of x and y are M and D respectively, if m, the intrinsic dimension of y is such that $m \leq M$, then the latent variable z must have dimension $K = D - m$, assuming that the intrinsic and nominal dimensions of x are the same. In the case that $M + K > D$, the solution is to create a vector $x_0 \in \mathbb{R}^{M+K-D}$ of zeros and concatenate it to x.

In order to improve training efficiency, INN allows to simultaneously optimize for losses on the input and output domain. It is performed forward and backward iteration in an alternating way accumulating gradients before updating the parameters. In the forward iteration, it is penalized the difference between the simulation outcome and the predicted value by the network with some loss function $L_y(y, f_y(x))$.

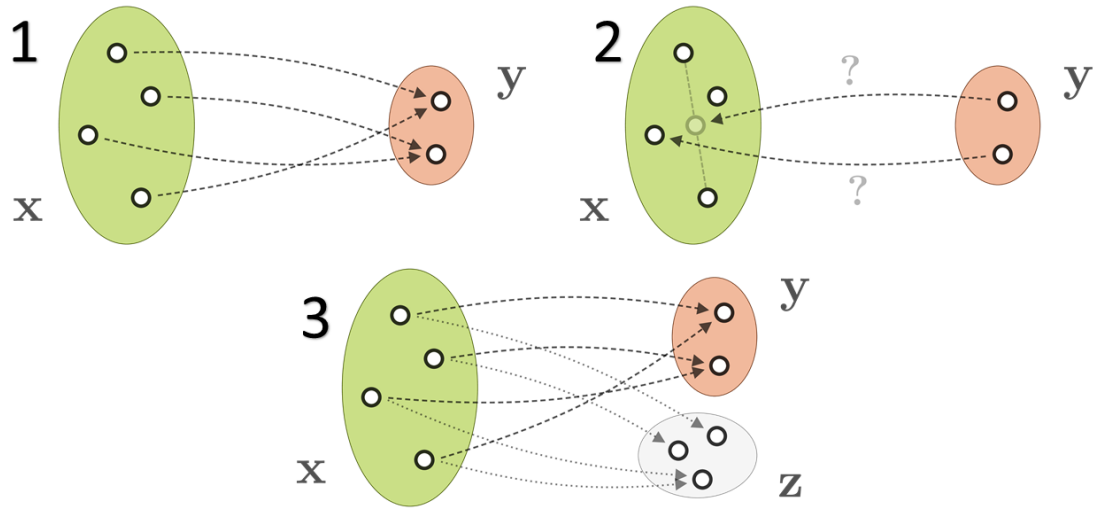


Figure 3.8: 1: Ambiguity on the forward model where **x** are mapped onto identical observations **y**, 2: Inherited information loss on the inverse model, 3: INN approach with latent variable z

The loss for latent variables penalizes the mismatch between the joint distribution of network outputs $q(y = f_y(x), z = f_z(x))$ and the product of marginal distributions of simulation outcomes $p(y = s(x))$ and latent $p(z)$ as $L_z(q(y,z), p(y)p(z))$ [25]. By locking the gradients of L_z with respect to y it is guaranteed that the following updates only affect the new predictions on z and do not worsen the ones on y. To speed up convergence, a loss L_x on the input side is defined which penalizes deviations between the distribution of backward prediction $q(x)$ and anterior data distribution $p(x)$.

Even though we believe this network can be suitable for the nonlinear inverse problems this thesis is trying to solve, there are some disadvantages compared to the MLP approach. INN main application is on ambiguous inverse functions [26], and as these thesis problems are supposed to satisfy the existence and uniqueness conditions, the ambiguity this architecture is trying to combat is non-existing, making it an out of proportion approach. Another downside is the fact that there are no viable frameworks or libraries available on this architecture, therefore it would be required to create it from the ground up, which is not the aim of this thesis. For the aforementioned reasons, the application of this architecture to solve the proposed problem is conditioned to the available time, not being the main approach.

3.4 Ocean Color

We aim to apply the developed technique to the Ocean Color (OC) research field, which studies water and its constituents through the reflectance of the sunlight out of the ocean. With the assistance of satellites, we are able to detect spectral variations in the water leaving radiance which is affected by multiple ocean compounds. In the open ocean, the signal is primarily dependent on phytoplankton which contain photosynthetic pigments, primarily chlorophyll-a (Chla) and an assemblage of other pigments, and which coexist together with colored dissolved organic matter (CDOM) that are related to the phytoplankton [31]. The optically active seawater compounds can be estimated based on the relation between them and the remote sensing measurements of sea-surface reflectance.

Inherent Optical Properties (IOPs) of the water quantify such relation in terms of scattering and absorption values. The light fraction that ultimately leaves the sea surface and can be measured from space-borne sensors (after correcting for the atmosphere contribution) is a function of these IOP values. In [5], the relation between the remote sensing reflectance (R_{rs}) at a certain wavelength (λ) and the IOPs (absorption a, and backscattering b_b) is modeled as in Figure 3.9.

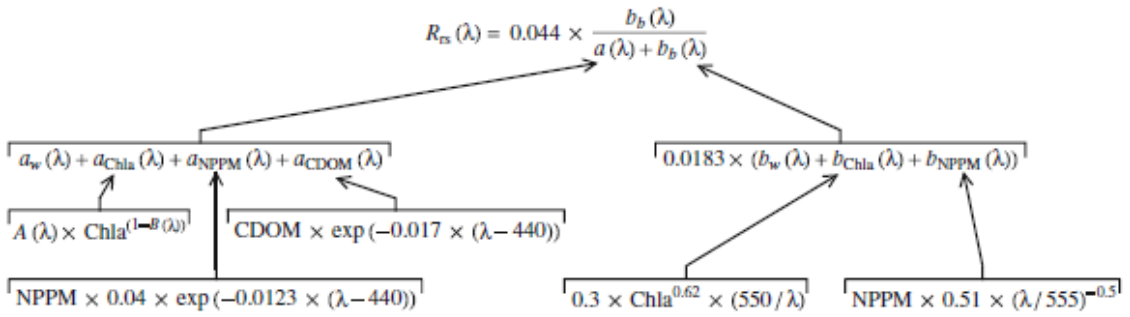


Figure 3.9: The forward model is a function from the optically active seawater compounds(Chla, NPPM and CDOM) to the remote sensing reflectance (R_{rs}) at a given wavelength(λ)

In this thesis, the OC compounds we aim to estimate are: the concentration of chlorophyll-a and phaeopigments (Chla), the concentration of non-pigmented particles (NPPM), and the colored dissolved organic matter (CDOM).

The total absorption, a, results from the additive contribution of the absorption of seawater (a_w), phytoplankton (a_{Chla}), non-pigmented particulate matter (a_{NPPM}) and colored dissolved organic matter (a_{CDOM}) at a given wavelength (λ). Similarly the total backscattering, b_b , is given by the additive contribution of the backscattering of seawater (b_w), phytoplankton (b_{Chla}) and non-pigmented particulate matter (b_{NPPM}) at a given wavelength (λ).

The colored dissolved organic matter does not produce any light scattering effect. The $a_w(\lambda)$ and $b_w(\lambda)$ are constant for a given wavelength. The $A(\lambda)$ and $B(\lambda)$ constants parameterize the phytoplankton chlorophyll-specific absorption coefficient.

3.4.1 Forward Model

In this thesis, we are going to follow the study in [19] where 6 different wavelengths were considered. These correspond to the central bandwidths of the SeaWiFS [32] sensor. The considered wavelengths λ and the respective values for $a_w(\lambda), b_w(\lambda), A(\lambda)$ and $B(\lambda)$ [19, 33] are presented in Table 3.2.

λ	$a_w(\lambda)$	$b_w(\lambda)$	$\mathbf{A}(\lambda)$	$\mathbf{B}(\lambda)$
412	0.00544	0.004679	0.03224	0.2870
443	0.00902	0.003479	0.03935	0.3435
490	0.01850	0.002300	0.02726	0.3616
510	0.03820	0.001900	0.01804	0.2618
555	0.06900	0.001400	0.00703	0.0307
670	0.43460	0.000600	0.01848	0.1478

Table 3.2: Matrix of coefficients for wavelength-dependent values

Representing the OC products Chla, NPPM and CDOM by variables m_1, m_2, m_3 and the observations of reflectance at 412, 443, 490, 510, 555, 670 nm by variables $o_1, ..., o_6$, the above forward model becomes (3.1):

$$o_i = f_i(m_1, m_2, m_3) = \frac{0.044}{\dfrac{a_w(\lambda_i) + A(\lambda_i) m_1^{1-B(\lambda_i)} + e^{-0.017(\lambda_i - 440)} m_3 + 0.04 e^{-0.0123(\lambda_i - 440)} m_2}{0.0183(b_w(\lambda_i) + 0.3(550/\lambda_i) m_1^{0.62} + 0.51(\lambda_i/555)^{-0.5} m_2)} + 1} + \epsilon_i \tag{3.1}$$

where ϵ_i is the measurement error committed at the respective wavelength λ_i.

3.4.2 Parameters

The geophysical parameters have been measured to great extent in [34], and a simplified version containing the products that are related to this thesis is presented in Figure 3.10.

Following [19] and accordingly to [35] we will assume Gaussian measurement error distributions p_i (3.2) with mean $\mu_i = 0$ and where standard deviation, σ_i, is 5% of the obtained measurement o_i (except for λ_6 in which it is 6% of o_6):

$$p_i(\varepsilon_i) = \frac{1}{\sigma_i \sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\varepsilon_i}{\sigma_i}\right)^2} \quad \text{with} \quad \sigma_i = \begin{cases} 0.05 \times o_i, & 1 \le i \le 5 \\ 0.06 \times o_i, & i = 6 \end{cases} \tag{3.2}$$

With such assumptions, we will be able to generate and test data accordingly since we have both the domains of the model and the observable parameters and an apriori distribution of the measuring errors.

Geophysical Parameter	Geophysical Range	Comments
Normalized water-leaving radiances	0 - 10 mW cm^{-2} μm^{-1} sr^{-1}	Wavelength dependent
Remote sensing reflectances	0 - 0.08 sr^{-1}	Wavelength dependent
Chlorophyll-a	0 - 500 mg m^{-3}	
Diffuse attenuation coefficient	0.02 - 8.0 m^{-1}	Heritage missions focused on Kd(490)
Inherent optical properties: - Absorption coefficient	0.02 - 2 m^{-1}	
- Backscatter coeff. (bb)	0.0003 - 0.1 m^{-1}	Wavelength dependent. Specific ranges for absorption can be subdivided into phytoplankton, detrital (or perhaps "depigmented or bleached SPM") and CDOM.
- Beam-c	0.03 - 10 m^{-1}	
Particulate inorganic carbon	0.000012 - 0.00053 mol m^{-3}	
Particulate organic carbon	15 - 2000 mg m^{-3}	POC can reach nearly 3000 mg m^{-3} in Chesapeake Bay and even higher in rivers throughout the globe.
Dissolved organic carbon	35 - 800 µmol C l^{-1}	Such high values are only found in rivers. Estuarine values generally do not exceed 500 µmol C l^{-1}
Coloured dissolved organic matter (also known as yellow substance, and gelbstoff), bleached particle absorption	0.002-0.9 m^{-1}	CDOM is not quantified in the same way as DOC or Chl-a. One approach is to measure CDOM fluorescence (UV excitation & blue emission) and scale response to the concentration of quinine sulfate for the same fluorescence response.

Figure 3.10: Range of observed geophysical parameter values. The geophysical ranges were determined after an extensive literature survey and data analyses by the Aerosol, Cloud, Ecology (ACE) mission ocean working group.

3.4.3 Validation

To validate our approach we will compare it to the work in [19], we have access to their obtained results on 12 simulated experiments representative of the different seawater types that can be found in nature. For each test, the model parameters were established and the values of the observed parameters were generated with the forward model (Table 3.3). Such simulated values were used instead of real measurements allowing the comparison of the estimated parameter values with the exact values used to simulate the observations.

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
Chla	1	5	10	0	0	0	0	0	0	0.1	1	10
N P P M	0.374	1.141	1.843	0.5	1	5	0.5	0.5	0.5	0.1	0.5	2
C D O M	0.012	0.035	0.055	0	0	0	0.1	0.5	2	0.5	5	0.5

Table 3.3: The 12 experimental cases extracted from [19]

4

Technical Approach

4.1 Data

The main objective of this dissertation is to be able to compute the inverse of a given function f. This function f is a parameter provided by the user which is used to generate a dataset of pairs (M, O) with M being the model parameters and O the observed parameters produced by the computation of $f(M)$, this function will also be used in order to test the possibility of training the network with regard to the forward model.

For this thesis context, the M parameters domain is going to be restrained to a margin that contains the values that are attainable in the real world.

4.1.1 Data Generation

As in this thesis, we are not working with a fixed pre-existing dataset, instead, we aim to employ the forward model to generate any sort of necessary data. For this, it is essential that the user provides two main parameters: the forward model (f) whose inverse is aimed to compute, and the bounds for each model parameter m_i. It should be obvious that without f it is impossible to generate any dataset as the transformations applied to the model parameters to obtain the observed ones are unknown, but less obvious is the necessity to have the bounds for each model parameter. Even though the study of the whole domain of the inverse function is a valuable task, it is not possible to generate a dataset that is able to train the network when its domain is endless. Therefore, we'll limit the domain of the model parameters to a domain that includes a feasible space in the real world, allowing the network to master it and be able to provide accurate predictions of the observed parameters.

Creating each example k involves two main steps, generating a uniformly distributed set of model parameters $M^k = (m_1^k, m_2^k, ..., m_n^k)$ that are within the user-defined bounds (being Lm the upper bound and lm lower bound), and apply those model parameter to the forward model in order to compute O^k.

$$m_i^k = U(0,1) \times |Lm_i - lm_i| + lm_i$$

$$O^k = f(M^k)$$

To implement this, and most of the mathematical work developed in this thesis we resorted to the NumPy library. NumPy is a Python library that supports the use of multi-dimensional arrays and matrices, along with a platter of high-level mathematical functions to operate on these arrays.

With this formula we are able to develop any dataset capable of training, validating, and testing our model. Some scaling may be performed on this data in order to improve the network training.

4.1.2 Re-scaling the Data

As stated in [36], re-scaling the data is beneficial to improve the numerical stability of the model and often reduces training time.

Having the gradient descent formula presented in 2.1. The step size of the gradient descent will be affected by the presence of feature value X in the formula. Because of the differences in feature ranges, each feature will have a different step size. Scaling the data before feeding it into the model guarantees that the gradient descent moves smoothly towards the minima and that the gradient descent steps are updated at the same rate for all features. This means that unscaled input variables can result in a slow or unstable learning process, whereas unscaled target variables on regression problems can result in exploding gradients causing the learning process to fail. Having features that are on the same scale helps speed up the gradient drop to the minimum.

In this thesis, we will apply two types of data re-scaling: Min-Max Normalization and Standardization.

The first method rescales the range of the data to $[0,1]$ by applying the following formula:

$$X_i' = \frac{X_i - X_{min}}{X_{max} - X_{min}}$$

Where X_{min} and X_{max} are the minimum and the maximum values of each feature.

The other scaling technique focus on centering the values around the mean with a unit standard deviation. Standardization makes it so that the mean of the attribute becomes zero and the resultant distribution has a unit standard deviation.

$$X_i' = \frac{X_i - \mu(X_i)}{\sigma(X_i)}$$

The choice of using normalization or standardization depends on the problem and machine learning algorithm that's been used. However, the consensus is that:

- Normalization is good to use when you know that the distribution of your data does not follow a Gaussian distribution.
- Standardization is preferable in cases where the data follows a Gaussian distribution. However, this does not have to be necessarily true. Also, unlike normalization,

standardization does not have a bounding range. So, even if you have outliers in your data, they will not be affected by standardization.

- It is a good practice to fit the scalar on the training data and then uses it to transform the testing data. This would avoid any data leakage during the model testing process. Also, the scaling of target values is generally not required.

Having this "rules" in mind, we believe that the best process of scaling the model parameters is Normalization, because, as previously explained, the dataset follows a uniform distribution on the model parameters, and the min-max bounds are already known as they are presented by the user. However as stated in the last point, it is not required to scale the target values and therefore we will analyze the difference between scaling and not scaling these parameters (extended in 4.2.2.1). The best scaling technique of the observed parameters may vary from problem to problem. Min-Max Normalization can work great if the function produces uniformly distributed values however, most likely, Standardization works best as outliers do not have as much effect on the scale therefore do not "condense" the other values as much.

These hypotheses are evaluated in the oncoming chapter.

4.1.3 Splitting the Data

In order to better evaluate the performance of the model, we will apply an approximation of the training-validation-testing technique. The procedure involves taking a dataset and dividing it into three subsets. The first subset is used to fit the model and is referred to as the training dataset. The second set, the validation one, is used in order to obtain the error estimates to select the best hypothesis. The third subset is not used to train the model instead, the input element of the dataset is provided to the model, then predictions are made and compared to the expected values. The objective is to estimate the performance of the neural network on new data: data not used to train the model. This is how we expect to use the model in practice. Namely, to fit it on available data with known inputs and outputs, then make predictions on new examples in the future where we do not have the expected output or target values.

As in this thesis, we are studying the application of an adaptive data generator 4.4, the training set will be constantly altered, however, the validation and testing data will stay the same for the whole computation as the evaluation of the model are independent of these sets and therefore their re-computation is superfluous.

This procedure has one main configuration parameter, which is the size of the train, validation, and test sets. As there is no optimal value here and we are able to generate any number of points to create a dataset, we will allow the user to define the size of each subset, having into consideration that larger sets increase performance by the cost of runtime.

4.2 Model

As presented in 2.1.1, we will be using a MLP as our neural network architecture. We want to compute $f^{-1} : O \rightarrow M$, to do so, we will be using a regression model that is able to map this function. A regression algorithm is one that can predict the values of dependent variables based on the values of independent variables. In order to achieve this, the network is fed with the previously created training dataset and allowed to train for a finite number of epochs.

This section provides some enlightenment on the initialization parameters of the model and some practices that will be explored by this thesis.

4.2.1 Layers and Neurons

There is no optimal way of choosing the number of layers and the number of nodes (neurons) in each layer as each model will behave differently for each problem depending on the complexity of the latest.

However by following a small set of clear rules, one can set a competent network architecture. Following this schema will allow the user to develop a competent architecture but probably not an optimal one, which once this network is initialized, the user can iteratively tune the configuration during training using a number of ancillary algorithms like for example pruning nodes based on values of the weight vector after a certain number of training epochs.

As previously explained in Section 2.1.1, every MLP is composed of three types of layers: input, hidden, and output. Determining values for the number of layers of each type and the number of nodes in each of these layers is necessary to create the network architecture.

4.2.1.1 Input Layer

The amount of input layers is quite simple, every network has exactly one of them. With respect to the number of neurons comprising this layer, this parameter is completely and uniquely determined by the shape of the training data, more specifically, the number of neurons comprising that layer is equal to the number of features, observed parameters in this thesis context, in the user's data.

4.2.1.2 Output Layer

Similarly to the input layer, each model has exactly one output layer, and its size is also determined by the shape of the training data. The number of neurons comprising this layer is the same as the number of labels, and model parameters in this thesis context, in the user's data.

4.2.1.3 Hidden Layer

In the literature, there is proof that a single hidden layer neural network is capable of universal approximation. The universal approximation theorem states that a feed-forward network, like MLP, with a single hidden layer, containing a finite number of neurons, can approximate continuous functions with some assumptions on the activation function. However, it does not specify how easy or how long it will take for that model to actually learn the complexity of the data. Similarly, there is no magical number of neurons for each layer that provides the best network learning accuracy, it is the user that has to adapt the model for the needs of the problem.

A simple methodology to ensure that the model is able to learn the pattern without overfitting the data is to start with a simpler model and gradually increase the network complexity if during training the network does not show any improvements. Once the network starts to overfit the training data the user can either scale down the network or apply some regularization methods to the model.

4.2.2 Activation function

The chosen activation function to be used in the hidden layers of our model is a variation of ReLU, the LeakyReLU function.

$$LeakyReLU(x) = \begin{cases} 0.3x & \text{if} \quad x < 0 \\ x & \text{if} \quad x >= 0 \end{cases}$$

The choice of ReLUs over the other functions mentioned in Section 2.1.2 can be justified by three main advantages: **computational speed**, **vanishing gradients** and **convergence speed**.

ReLUs are simpler to compute. Both the forward and the backward pass through ReLU are just simple conditional statements that either returns their own value or a linear computation of it, unlike the Sigmoid and Tanh activation, in comparison, that require computing at least one exponent. This advantage is huge when dealing with big networks with many neurons, and can significantly reduce both training and evaluation times.

Another downside of these functions is how easy they saturate. The range of inputs where the Sigmoid and Tanh's derivative is sufficiently nonzero is relatively small. This means that making a backward pass through these once it reaches the left or right plateau is nearly pointless because the derivative is so close to 0. On the other hand, the ReLU derivative only is 0 for negative values of input and even this can be eliminated when using leaky ReLUs (Figure 4.1).

Finally, with ReLu activation, the gradient of the ReLu is either 0 or 1 (0.3 or 1 in the case of LeakyReLU) which means that often, after many layers, the gradient will include the product of 1's and the overall gradient won't be too small or too large. On the other end, the gradients of Sigmoid and Tanh are typically some fraction between 0 and 1,

which when working with a large number of layers, multiply and result on an overall gradient that is exponentially small, so each step of gradient descent will make only a small change to the weights, leading to slow convergence.

Having this in consideration, we are certain that training with ReLU, more specifically LeakyReLU will provide us with better model performance and faster convergence. As ReLUs have been proven to have some issues with Glorot Initialization [37], we will be using the current standard approach for weight initialization for neural network layers with this activation function, the He initialization (Section 2.1.3).

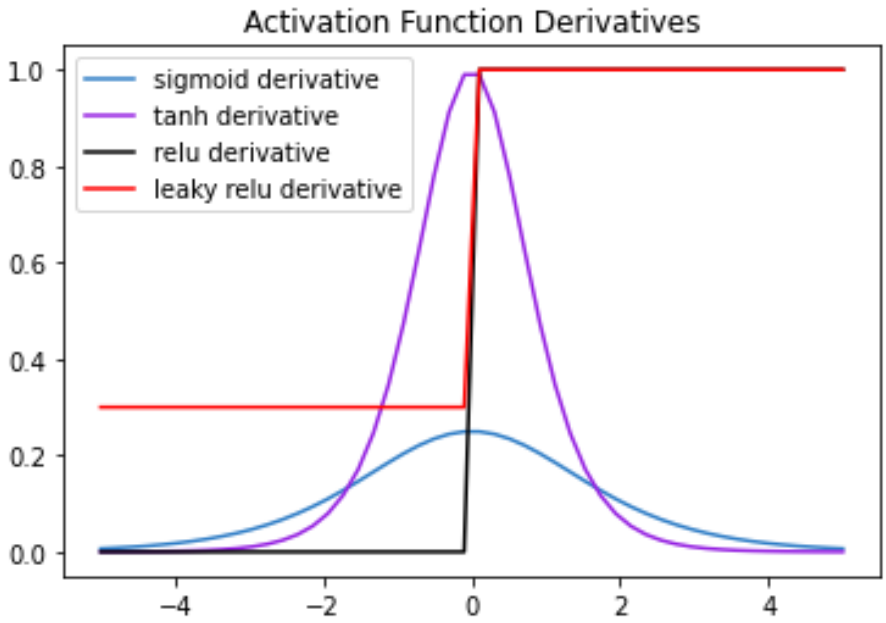


Figure 4.1: Derivatives of diverse activation functions

4.2.2.1 Output Function

Unlike in the hidden layers, we are not so certain which activation function to use in the output layer to guarantee the best results. In this thesis, we will be studying if bounding the predictions of the network to the limits presented by the user allows for greater accuracy of the network or if allowing it to learn past these boundaries is preferable.

These two test scenarios depend not only on the activation function used on the output layer but also on the scaling of the model parameters presented in Section 4.1.2.

In the first test scenario, we will evaluate the performance of the model when having an unrestricted domain of outputs. To do so, we will resort to the use of the unbounded linear activation function which allows the model to make predictions in the $[-\infty, \infty]$ domain. This test will be divided into 3 subtests where it also evaluates if scaling the model parameters results in a better performance compared to unscaled parameters.

The second test scenario is to evaluate the performance of a model that has a bounded domain of predictions. The two functions that suppress the input value to a bounded range are the sigmoid and tanh functions. These two functions are quite similar, where the main difference is the behavior of their gradient. As can be seen in Figure 4.1, the gradient of tanh is four times greater than the gradient of the sigmoid function. This means that using the tanh activation function results in higher values of gradient during training therefore higher updates in the weights of the network.

One detail we must have in consideration is that these functions are bounded to $[0,1]$ in the case of the sigmoid and $[-1,1]$ for tanh. It is required to ensure that the scale of the output variables matches the scale of the used activation function on the output layer. To do so we can resort to the Min-Max normalization process of data re-scaling, this would scale the data perfectly to the $[0,1]$ bounds or with some easy additional computations $(O'' = (O' \times 2) - 1)$ to the $[-1,1]$ bounds. However, it is to notice that these functions only predict close to their boundaries when their input is close to or equal to ∞ (or $-\infty$), making it nearly impossible for the model to learn the boundaries. A simple solution to overcome this problem is to instead of exactly matching the user boundaries with the activation boundaries to allow for a small margin α, therefore scaling the data within $[0+\alpha, 1-\alpha]$ for the sigmoid function or $[-1+\alpha, 1-\alpha]$ for the tanh function. In this thesis we will consider $\alpha = 0.05$.

Once the network has finished training and we aim to acquire the values of the actual prediction, it is required to apply the inverse of the transformation made as the outputs will also be bounded to the limits of the activation function.

4.2.3 Loss Function

In this thesis we will be analysing the application of the three most common loss functions when it comes to regression problems: MSE, MAE and Huber (Section 2.1.5). In addition to these losses we also want to test the hypothesis that adding the forward model in the error function may allow the network to more easily converge to a minimum.

4.2.3.1 Mean Squared Error with regard to the Forward Model

In order to study if the inclusion of the forward model into the measurement of the performance has any impact on efficiency, a new loss function with regard to the model was developed. This new function computes the mean squared difference between the observed parameters predicted by the network and the true value we expected. This is achieved by applying the forward model to the network predictions when computing the loss function.

$$MSEw/FM(\Theta) = \frac{1}{N}\sum_{i=1}^{N}(f(m_i) - f(\hat{m}_i))^2 = \frac{1}{N}\sum_{i=1}^{N}(o_i - \hat{o}_i)^2$$

Being m and $\hat{m}$ the true value and model prediction for the model parameter respectively, and f a TensorFlow function provided by the user that mimics the transformations of the forward model (see Annex I for a tool on how to create this function), the computation begins by applying the inverse of the scaling procedures on the model parameters, as computing the observed parameters using the forward model on scaled parameters will have different results or may even be unfeasible. Once computed the model's observed prediction ($\hat{o}$) and the truly observed parameters (o), the MSE can be applied to those values.

Even though this method requires additional computations and therefore will be slower than some of the other loss functions, it may produce better conversions to a minimum and in some cases provide better gradients during learning.

4.3 Call Backs

Another major challenge when dealing with neural networks is how to train them in order to maximize accuracy. When training a network, there is a point where the model stops learning the inverse mapping of the function and starts to over-adapt to the details of the training set that do not generalize from the data that were generated from the forward model. This will increase the error, making the model less useful for making predictions on new data. The challenge is to train the network long enough that it is capable of learning the inverse of the given function but not training the model so long that it overfits the training data. In this section, we aim to present some of the techniques that will be used in order to mitigate overfitting the training data while maximizing the accuracy of the network.

4.3.1 Early Stopping

One way to prevent overfitting is not allowing the network to hit this aforementioned turning point where the model starts to overly adapt to the details of the training set.

Early Stopping achieves this by evaluating the model during training on a validation dataset after each epoch. If the performance of the model on the validation dataset starts to degrade, then the training process is stopped.

In order to implement Early Stopping the network must be under-constrained, meaning that it has more capacity than is required for the problem. It's also required to have performance metrics and a trigger to stop training.

Under constraining the network is achieved by presetting the number of training epochs to a larger value than what may normally be required, therefore having enough opportunity to learn the training dataset and to begin to overfit to it.

Once the training starts the model is periodically evaluated with some performance metrics, the loss on a validation dataset is the most commonly used metric to monitor and the one used in this thesis. The periodicity of the evaluation adds an additional

computational cost during training, setting it to every few epochs can reduce training time.

Once a scheme for evaluating the model is selected, a trigger for stopping the training process must be chosen. The trigger will use the monitored performance metric to decide when to stop training.

4.3.2 Reducing Learning Rate

Besides combating overfitting we also want the model to be able to maximize predictions. One common occurrence is when the model gets "stuck" in a region similar to the one represented in Figure 4.2, in this situation, the weight updates would be endlessly jumping over the minima without any major updates. In this situation, the model training stagnates and not even the optimizer learning step update is able to adapt.

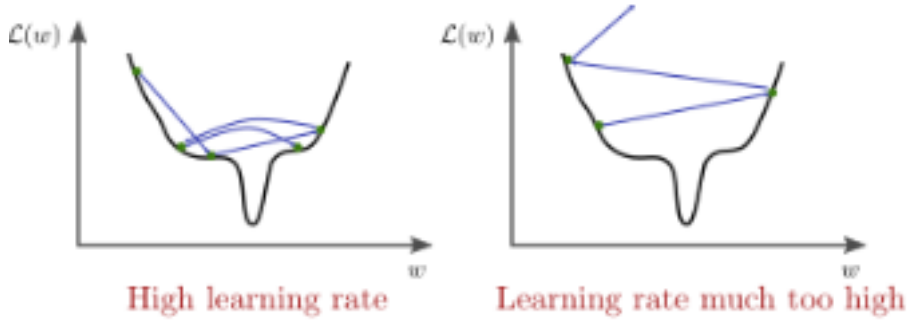


Figure 4.2: Loss Surface [38]

The solution here is to "manually" reduce the learning rate. The Keras API provides a ReduceLROnPlateau class, that decreases the learning rate by a user-defined decay when the given metric is stagnating for longer than the patience allowed. For the context of this thesis, we reduce the learning rate by a factor of 0.1 if the training loss does not decrease past 5 consecutive epochs.

4.4 Adaptive Data Generator

Unlike most problems that involve machine learning, this thesis has the particularity that we possess a smilingly endless dataset in the form of a forward model. Therefore, what we aim to study is the application of a data generator that computes new data so that we are able to further increase the performance of the network by continuously creating it with more data.

To do so we resorted to a Keras module which is able to during the training phase, generate data in parallel by the CPU and then directly fed it to the GPU to fit the model. In order to implement this, it is needed to define a class inherited from TensorFlow.Keras.utils.Sequence and define the initializer, the number of batches per epoch,

and how to generate the data. In addition, it can be defined a method on_epoch_end, which is executed at the end of each epoch.

The class initializer is how we introduce data into the class. Here we pass the user-defined limits for each model parameter, the forward model, the scaling factors for data re-scaling (e.g.: mean values, standard deviation, minimum and maximum values of each parameter), the dataset size, and the batch size.

Each data generation call requests a batch index between 0 and the total number of batches, where the latter is specified by this simple value $\frac{dataset\_size}{batch\_size}$, in this thesis we defaulted the batch size to the Keras default value of 32.

Now, when the batch corresponding to a given index is called, the generator executes the method to generate the data for that batch. This allows us to compute a continuous dataset and study a larger domain compared to the traditional method of having a fixed dataset as each batch and each epoch has different training data than the one before. In this thesis, we also want to study the effects of an adaptive training generator so we will compare three data generating techniques: **random sampling**, **generate model parameters** with the most error, **generate observed parameters** with the most error.

4.4.1 Random Sampling

This method consists in simply generating a large dataset with the approach explained in Section 4.1.1, it does not have into consideration the model's training loss metrics and remains unaltered throughout training. This naive approach will be the basis of the performance comparison of the following two more advanced techniques.

4.4.2 Generate Model Parameters

With this technique, we aim to generate data points where the network had the highest error in the previous epoch. Similarly to the previous approach, the first epoch of training will be done with randomly generated points, however from there, we will generate new points in regard to the training error on the previous epoch, by creating the aforementioned on_epoch_end method.

This method begins by computing the probability density function of each point of the model parameter domain in regard to the error. To achieve this we resorted to the Kernel Density Estimation (KDE) algorithm, this algorithm is very useful as not only allows us to visualize the data as a kind of continuous replacement for the discrete histogram as it can also be used to generate points that look like they came from a certain dataset, which in this case are the points where the model has lower accuracy and presenting the network new training data where it previously performed poorly hopefully will allow it to adapt and increase its performance.

For our method we resorted to the weighted KDE function:

$$\hat{p}_n(v) = \frac{1}{nh} \sum_{i=1}^{n} w_i K\left(\frac{t - T_i}{h}\right)$$

Here T is the training dataset of the previous epoch and $K(x)$ is the kernel function that is generally a smooth, symmetric function, in our case it is used the Gaussian function:

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$$

h > 0 is the smoothing bandwidth that controls the amount of smoothing and w is a normalized weight vector that is comprised of the error of the model for each point T_i, computed using the mean squared error formula. Basically, this function smoothes each data point M_i into small density bumps with regard to its error and then sums all these small bumps together to obtain the final density estimate (Figure 4.3).

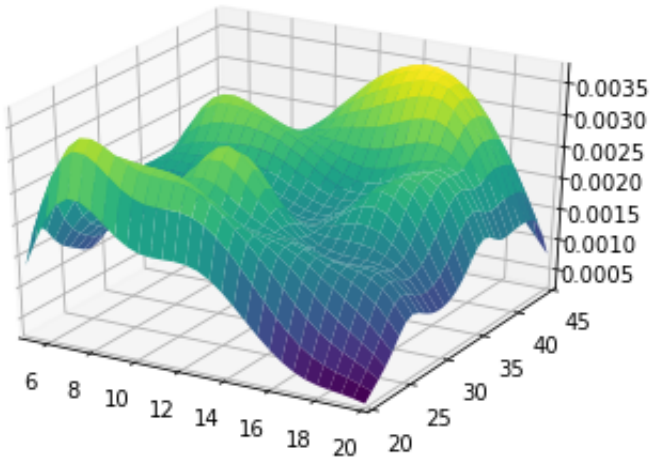


Figure 4.3: Error Distribution plotted by Kernel Density Estimation algorithm

In order to maintain the dataset stability while giving emphasis to the points where the network has the lowest accuracy it is not advisable to fully generate the batch from this distribution, instead, we will draw a percentage of the batch size, the default value is 10%, from this distribution and randomly compute the remainder of the dataset. In the case that the withdrawn sample exceeds the user-defined limits, the parameter(s) that exceed its value is rounded to the nearest limit. These samples are then applied to the forward model f in order to obtain the observed parameters.

4.4.3 Generate Observed Parameters

This technique is similar to the previous one. In the previous method, the error on the model parameters is used to plot a KDE however in this methodology the error on the observed parameters is used instead. Once the probability density function of the error in each point of the observed parameter domain is calculated, we are able to sample points of the observed parameters where the network has the lowest accuracy. However

with this approach the problem of computing the model parameters from the observed parameters this thesis is trying to solve presents again.

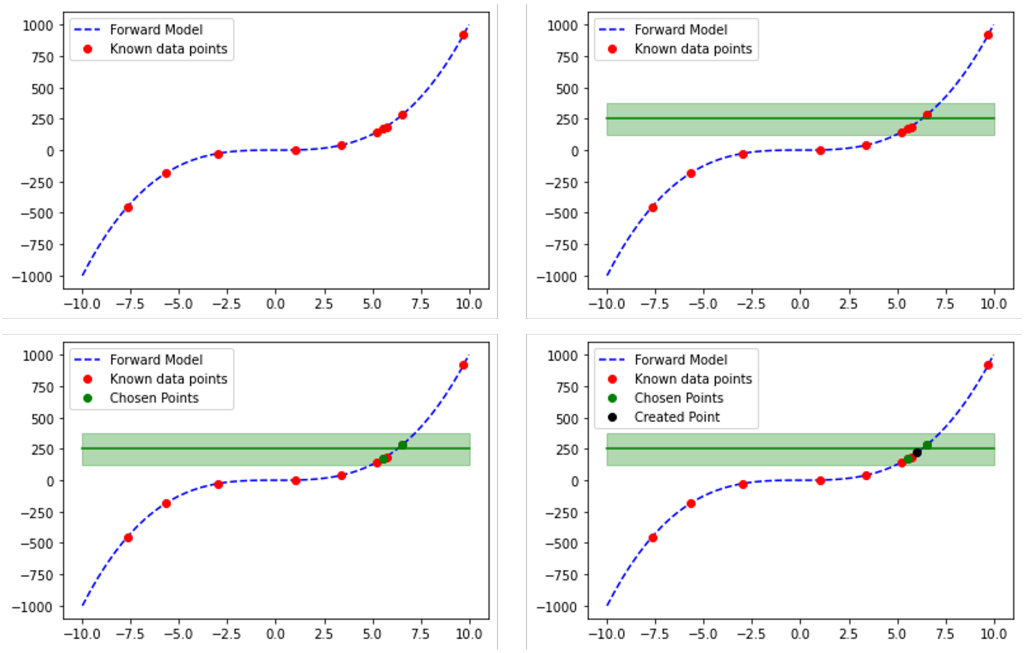


Figure 4.4: With an initial dataset equal to the top left image, we aim to compute a new data point whose observed value lies within 250 ± 125 (top right). The algorithm begins by selecting two random points (bottom left) with observed parameters in that range, in this case (5.526,168.74) and (6.534,279.06), and compute its model parameters midpoint which applied to the forward model results in a point within the desired range (6.03,219.30)(bottom right)

To solve this problem we developed an algorithm that combines the Stochastic Approach (3.2.3) and the SMOTE technique (2.2.2). Let us pretend that we aim to compute the model parameters M' of the sampled observed parameters O' given an absolute tolerance of α, this is if we find M' where $O' - \alpha \leq f(M') \leq O' + \alpha$ we accept $(M', f(M'))$ as our new point. Similarly to SMOTE, initially we search our existing dataset for two-point where its observed parameters lie in between $O' \pm \alpha$ and retrieve their model parameters, call them m_1 and m_2. As f is a continuous function there can be a set of model parameters between these two points that satisfy our search condition. So we sample m_3, a random point bounded by m_1 and m_2, and compute its observed parameters. If it satisfies $O' - \alpha \leq f(m_3) \leq O' + \alpha$ then this point is accepted as a new point for the dataset, otherwise, this point is stored in a dictionary and two new pairs of neighbors are used instead. In future iterations of this algorithm, the dictionary is searched before trying to compute any new point as it may already be found previously. An example of this algorithm at work can be seen in Figure 4.4.

4.5 Measuring Uncertainty

One question this thesis aims to answer is how to measure the effects of uncertainties on the model and measurement errors on the observable data. To do so we will resort to a Monte-Carlo technique where we sample multiple data points, add noise to them and see how the model predicts.

4.5.1 Noise Injection

It is physically impossible to perfectly measure any parameter as every measurement possesses an inherent uncertainty. We want to be able to represent this uncertainty on our data and therefore our training so that the network is still able to make valuable prediction having into consideration the measuring object error margin.

We will assume that the measurement error follows a Gaussian distributions p_i (4.1) with mean $\mu_i = 0$ and where standard deviation, σ_i, is equal to some percentage (e%) of the obtained measurement o_i:

$$p_i(\varepsilon_i) = \frac{1}{\sigma_i\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\varepsilon_i}{\sigma_i}\right)^2} \quad \text{with} \quad \sigma_i = \frac{e}{100} \times o_i \tag{4.1}$$

This "random" amount of error is then added to the observable parameters as noise in order to simulate the aforementioned measurement errors (Figure 4.5).

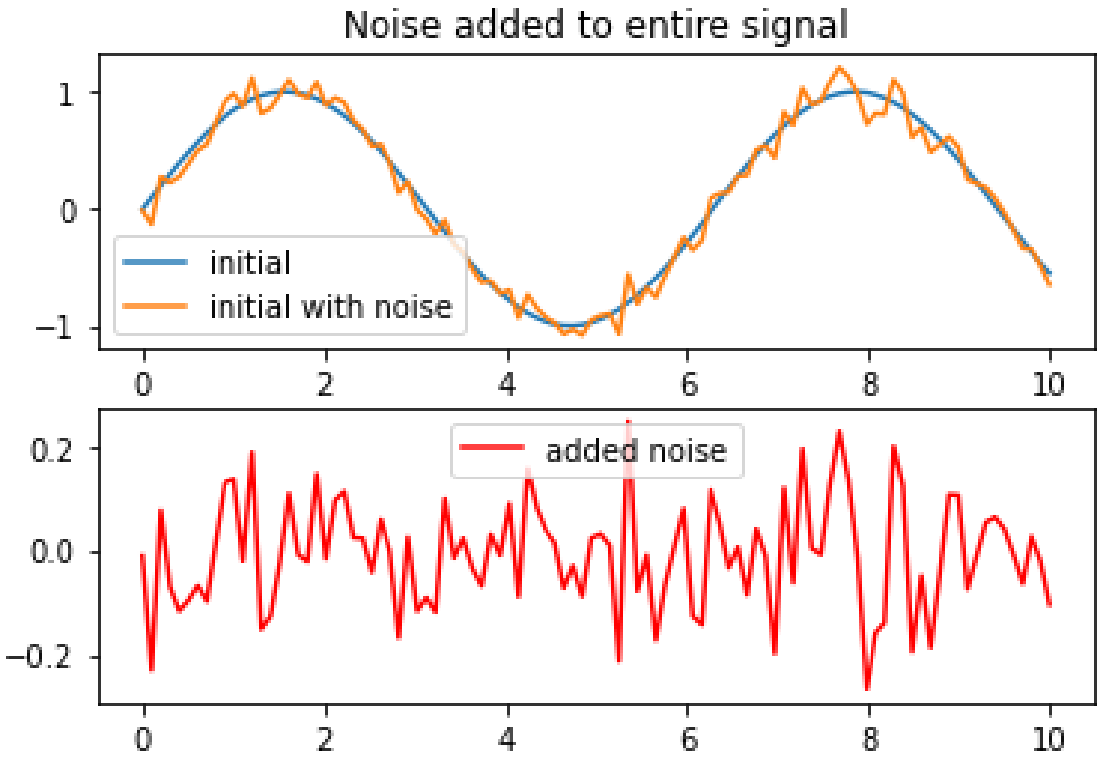


Figure 4.5: 15% Noise injection on Sin function

4.5.2 Monte-Carlo Technique

In order to measure the effects of uncertainty, either on measurements or approximations made in the model, we will resort to a Monte Carlo technique. A posteriori on our model training, our tests will take multiple instances of a fixed point (in the case of OC Problem the values on Table 3.3) and introduce noise into those points by the application of the aforementioned technique, transforming that initial value into a "cloud" of values. This cloud is then fed into the model for predictions. With the results from the model, we are able to plot the obtained joint distribution allowing us to identify regions of maximum likelihood as well as the possible error margin associated with our technique.

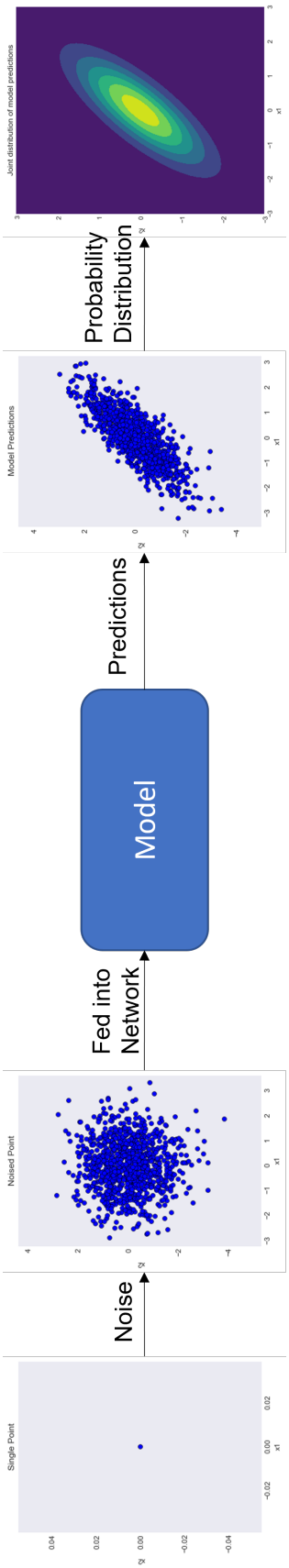


Figure 4.6: Monte Carlo Technique for validation

5

Results and Discussion

In this section, we present all the obtained results from the tests performed during the course of the realization of this dissertation. Initially, we will be evaluating a simpler non-linear problem, the projectile motion, and then the main problem this thesis aims to tackle, the Ocean Color problem. Finally we will provide a comparison analysis between this methodology and the Constraint Approach Method previously presented.

5.1 Projectile Motion

In order to focus on the broad characteristics of our method, without the complexity presented by the main problem, and to validate this approach among all of the inverse problem universe, we decided to begin our testing with a simpler, less computationally intensive problem, the projectile motion problem.

The forward model of this problem is a $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ function that through the model parameters of initial velocity (v_o) and launch angle (θ) is able to compute the observable parameters maximum height (h) and distance traveled (d) by the projectile as follows:

$$f(v_0, \theta) = (\frac{v_0^2 \times sin(\theta)^2}{2g}, \frac{v_o^2 \times sin(2\theta)}{g})$$

Where g is a constant of gravitational acceleration to which the value of 9.8 was assumed.

The domain of the model parameters was restrained to $v_0 \in [5, 20]$ and $\theta \in [20, 45]$, by doing so, it is asserted that the function is non-ambiguous in the entire domain of study. The inverse model we are trying to calculate is also a $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ mapping, that through the measured distance and altitude achieved by a projectile is able to compute the initial state of the launch.

5.1.1 Efficiency of Adaptive Data Generator

The efficiency of the data generator is related to its capability of generating new data points relevant to improve the network accuracy during training. In order to evaluate it, the three previously mentioned approaches were considered: **random sampling** (S_r), **model parameters sampling** (S_m) and **observed parameters sampling** (S_o). The test was performed using the same network, a simple 3-layer model with a linear output function and MSE as the loss function. The test evaluated the performance of the generator to create meaningful data over the course of 20 training epochs.

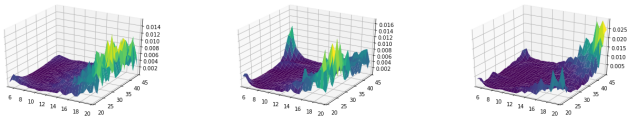
The naive approach of random sampling had the foreseen results, where the points of the network where the error was high would tend to stay high even after some epochs as there was no attempt of including these error metrics to create new data. This can be seen in Figure 5.1 where if the error is elevated for a certain domain of the model parameter, for example the upper limits of the model parameters as seen in 5.1c, tend to stay elevated even after training (5.1d and 5.1e).

The solution to improve training was therefore to generate data points in the domain where the network had the lowest accuracy. The solution implemented started the first 5 epochs with randomly sampled data points and from the 5^{th} epoch onwards it would increment the number of points drawn from the KDE algorithm, up to 10% of the batch size by epoch 20. As seen in Figure 5.2, this resulted in a more uniform error distribution (5.2e) and a better network performance.

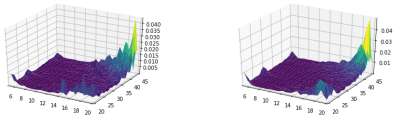
Lastly we evaluated the hypothesis of generating data where the error had the most error in regards to the observed parameters. This approach even though it converged to a better final result compared to the random sampling one, was significantly worse than the previous method (Table 5.1). This is due to the absolute tolerance parameter added in order to be able to compute this new points, as we are dealing with non-linear problems a small tolerance in the observed parameters may relate to a large discrepancy on the model parameters which results in generated point(s) that do not train the network as efficiently. Another downside of this approach is the ruining time. As can be seen in Figure 5.4, this approach has a linear time complexity while the previous approach possesses a constant time complexity, so, not only the previous approach provides better convergence, it is also faster and easier to implement. For this reasons, we believe that sampling data through the model parameters error is the best approach and the one that will be used on the next tests.

	S_r	S_m	S_o
MSE	4.016×10^{-2}	1.303×10^{-5}	6.783×10^{-3}
MAE	0.106	0.004	0.039

Table 5.1: Testing error for the three approaches after 20 epochs

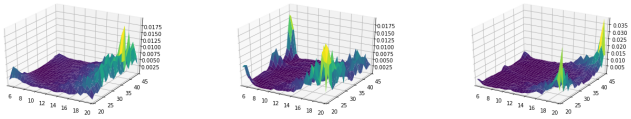


(a) Epoch 1 (b) Epoch 5 (c) Epoch 10

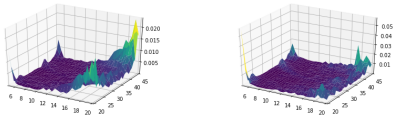


(d) Epoch 15 (e) Epoch 20

Figure 5.1: Error distribution for random sampling over the course of 20 epochs

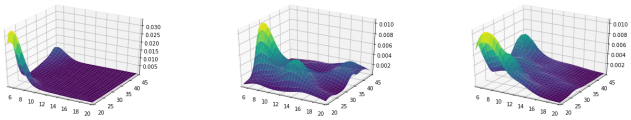


(a) Epoch 1 (b) Epoch 5 (c) Epoch 10

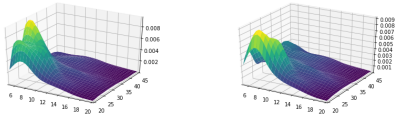


(d) Epoch 15 (e) Epoch 20

Figure 5.2: Error distribution for model sampling over the course of 20 epochs



(a) Epoch 1 (b) Epoch 5 (c) Epoch 10



(d) Epoch 15 (e) Epoch 20

Figure 5.3: Error distribution for observed sampling over the course of 20 epochs

Figure 5.4: Time comparison between model parameters sampling (in red) and observed parameters sampling (in green)

5.1.2 Evaluating Approaches

In this section, we will be testing which parameter combination provides the best training. The parameters in evaluation are:

- the scaling methods, which can be none, min-max normalization, and standardization;
- the loss function, we will be evaluating MSE, Huber Loss, and MSE with Forward Model;
- the output activation function: linear, sigmoid, or hyperbolic tangent functions;

Each training epoch will be composed of 10^5 data points, with a batch size of 32 points. The sampling technique used will be sampling data points where the network has the most error in regard to the model parameters, where the sampled points from the KDE algorithm is given by

$$\frac{e^x - e^{-x}}{e^x - e^{-x}} \times 0.1 \times batch\_size \qquad \text{where} \quad x = \frac{\#current\_epoch}{10}$$

this value is then rounded for the closest integer, the remanding points are randomly generated. This allows to generate up to 10% of each batch size from the error distribution which results in up to 10^4 data points from the KDE per epoch.

We will assume a network comprised of 3 layers, where the first two layers are static and the output layer varies according to the evaluated output function. Both layers will have the LeakyReLU activation function as presented in Section 4.2.2.

The training is complete when the validation loss of the network does not improve for 15 consecutive epochs.

The tests can be divided into two main categories: unbounded output function and bounded output function. In the unbounded output function category, the output layer will be composed of the linear output function, this section is composed of 27 tests, where every combination of model and observed parameter scaling is evaluated as well as the three loss functions.

The bounded output function category evaluates the performance of using sigmoid and tanh functions in the output layer. In this category the model parameters are always scaled to the bounds of the output function in evaluation, however, all 9 combinations of observed parameter scaling and loss function are studied for each output function.

In order to have a global metric system, we will be evaluating each approach on their descaled model parameters mean absolute error (True MAE) in an equal test dataset. The results are as presented in Table 5.2.

	Activation function	Model Parameters	Observed Parameters	MSE (True MAE)	Huber (True MAE)	MSE w Forward Model (True MAE)
Un-bounded	Linear	None	None	1,3032e-05 (0,0020)	2,9966e-05 (0,0044)	2,3155e-03 (0,5194)
			Min-Max Normalization	2,4971e-05 (0,0029)	7,4653e-06 (0,0023)	2,4344e-06 (12,5148)
			Standardized	4,9316e-05 (0,0039)	2,1897e-05 (0,0039)	2,0599e-05 (0,1602)
		Min-Max Normalization	None	6,3556e-08 (0,0035)	1,6116e-08 (0,0024)	4,5784e-03 (0,8214)
			Min-Max Normalization	1,3488e-07 (0,0047)	1,4502e-07 (0,0084)	2,5473e-06 (180,2655)
			Standardized	8,6386e-08 (0,0045)	1,0575e-07 (0,0067)	2,4069e-05 (0,1746)
		Standardized	None	7,7538e-07 (0,0034)	2,7365e-07 (0,0026)	4,5935e-03 (0,8229)
			Min-Max Normalization	1,8950e-06 (0,0063)	2,2018e-07 (0,0030)	2,6973e-06 (0,0858)
			Standardized	1,4909e-06 (0,0047)	1,4340e-06 (0,0062)	2,5881e-05 (0,1800)
Bounded	Sigmoid	Normalized Bounded [0,05,0,95]	None	7,3734e-07 (0,0096)	2,4670e-07 (0,0087)	4,7660e-03 (0,8913)
			Min-Max Normalization	1,5021e-06 (0,0156)	4,9013e-06 (0,0485)	3,5470e-06 (0,2637)
			Standardized	1,9463e-06 (0,0172)	6,1281e-07 (0,0177)	1,9122e-05 (0,1462)
	Tanh	Normalized Bounded [-0,95,0,95]	None	9,9621e-06 (0,0168)	3,1353e-06 (0,0147)	2,2266e-03 (0,6392)
			Min-Max Normalization	6,2738e-06 (0,0166)	3,1560e-06 (0,0150)	3,6816e-06 (0,2839)
			Standardized	6,2738e-06 (0,0166)	7,7414e-06 (0,0315)	2,1206e-05 (0,1883)

Table 5.2: Experimental results for Cannon Ball inverse problem

5.1.2.1 Loss function

By observing the results it is possible to see that even though the loss in regard to the Forward Model is able to train the model it provides poor results in comparison with the existing previous methodologies.

One major downside of this approach is that the predictions may fall outside the studied domain when using an unbounded output function and working with an ambiguous forward model. This can be seen in two of the previous tests where the model had a low value of loss but a high mean absolute error where in the first one the model predicted negative velocities and in the second angles between $[-160, -135]$ which even though are wrong, result in similar observed parameters. One example is when the model received as input the values of height and distance equal to 30.465625 and 5.091508 respectively which are the result of v_0 and θ being 17.975105 and 33.762418 degrees however the model predicted −17.976547 and 33.774433 which applied to the forward model wield a similar result to the input value of 30.475796 and 5.095519.

This happens due to the loss function itself being ambiguous and therefore depending on how the weights are initialized, it may be "cheaper" for the network to update the weights towards an ambiguous domain that is not the one in study (Figure 5.5). This problem was mitigated when a bounded function was used, as this disabled the possibility of predicting outside the studied domain, however, this technique kept having the worst performance when training.

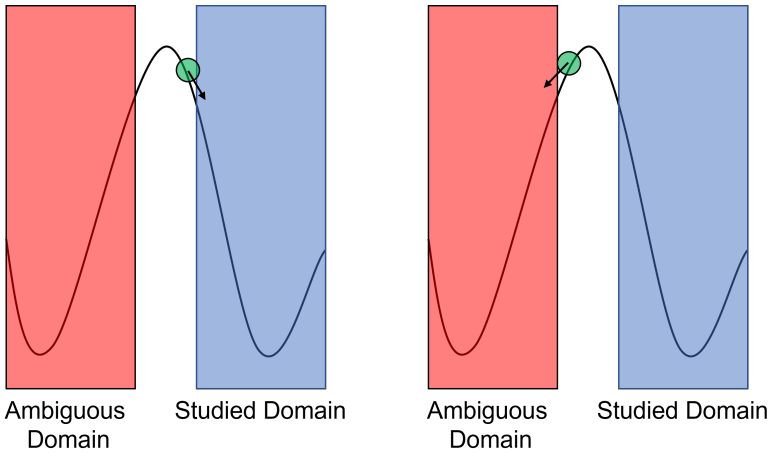


Figure 5.5: In black an example of an ambiguous loss function slope and in green the initial state of the network. In the first case, the network is initialized towards the studied domain and the training goes as planned. In the second situation, the model is initialized towards the ambiguous domain which has the same loss value however the predictions are not in the user-bounded domain

Another downside of this approach is the fact that it possesses a higher time complexity compared to the other functions, as it is required to de-scale the model predictions and apply them to the forward model which are additional computations before calculating the mean squared error.

5.1.2.2 Output Function

As can be seen above, the unbounded linear output function resulted at the minimum in 13% greater accuracy compared to the bounded output functions.

The linear output function managed to achieve a 4 times lower minimum MAE. It did not achieve the best mean MAE due to the 2 outliers where the predictions are not inaccurate as seen above but instead in the wrong domain and if those values are excluded it achieved a 0,1136 average value making it the best mean error. Excluding all the forward model loss tests, the linear model had on average a 475% increase in accuracy (Table 5.3).

	Minimum MAE	Average MAE	Average MAE w/o Forward Model Loss
Linear	0.002	3.1711	0.0042
Sigmoid	0.008	0.1576	0.0195
Tanh	0.015	0.1358	0.0185

Table 5.3: Minimum and Average absolute error for the studied output functions

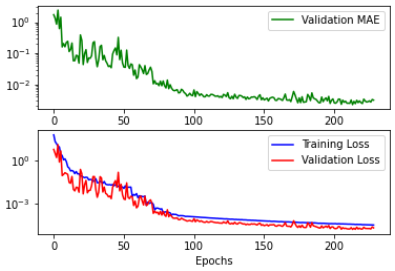
Having these results into consideration we believe that the unbounded linear output function ensures the best results when using deep learning to solve non-linear inverse problems.

5.1.2.3 Scaling the parameters

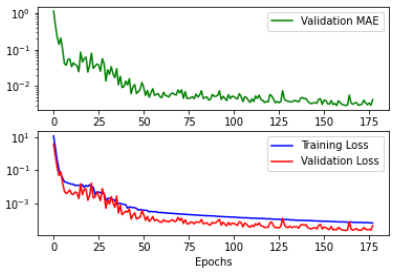
When it comes to parameter scaling we can conclude from the results above that applying transformations to the model parameters accrue in worse predictions compared to allowing the network to predict the original value.

When it comes to scaling the observed parameters, the accuracy of the predictions did not fluctuate as much from method to method. The main difference was that scaling provided an overall smoother and faster training when compared to no scaling (Figure 5.6 and 5.7).

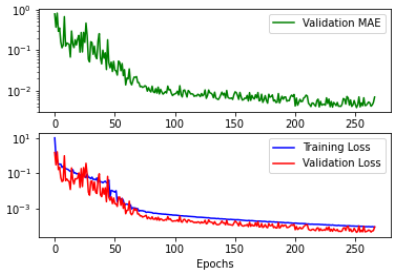
Having this considerations in mind, we believe that applying some data-scaling to the networks input data and no transformation to the output data has added value. However the data distribution and range is completely dependent on the problem at hands, therefore the user should study the correct transformations to apply in that specific case.



(a) No Scaling

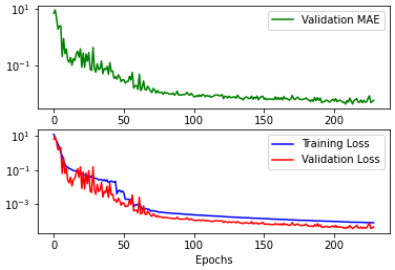


(b) Min-Max Normalization

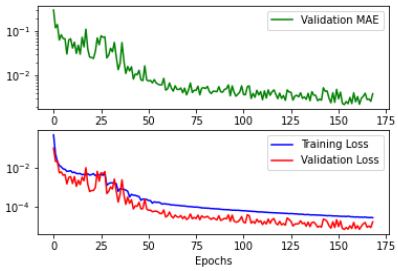


(c) Standardization

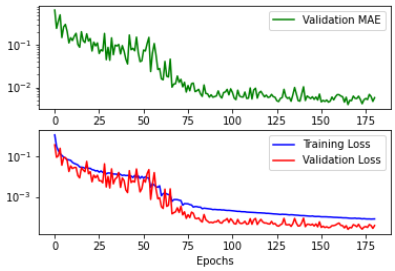
Figure 5.6: Training history for different observed parameter scaling with the linear output model, MSE loss function and no model parameter scaling



(a) No Scaling



(b) Min-Max Normalization



(c) Standardization

Figure 5.7: Training history for different observed parameter scaling with the linear output model, Huber loss function, and no model parameter scaling

5.2 Ocean Color

The ocean's color forward model is a $\mathbb{R}^3 \rightarrow \mathbb{R}^6$ that relates the measurements of the oceanic products of Chla, CDOM, and NPPM to observed reflectances at 6 different wavelengths (Equation 3.1). The domain of study used was extracted from [19], where through constraint propagation programming, they managed to compute an enclosure for the domain values where the function is not ambiguous and contains realistic values for the different seawater components. These values are $Chla \in [0,10]$, $CDOM \in [0,5]$ and $NPPM \in [0,5]$.

As in the previous problem we already discussed the broad regression methods to use for inverse problems, in this section, we will focus on the parameters intrinsic to this specific problem. We will use a linear output function, with no model parameter scaling, as in the literature and the previous results show that this combination provides the best results for regression problems. As a loss function, we will resort to MSE as in the previous tests it had the lowest minima and was more consistent overall. This section presents different studies such as:

- Best observed parameter scaling method for this problem
- Impact of sampling proportions on training
- Model development
- Impact of measurements uncertainty
- Validation of the methodology

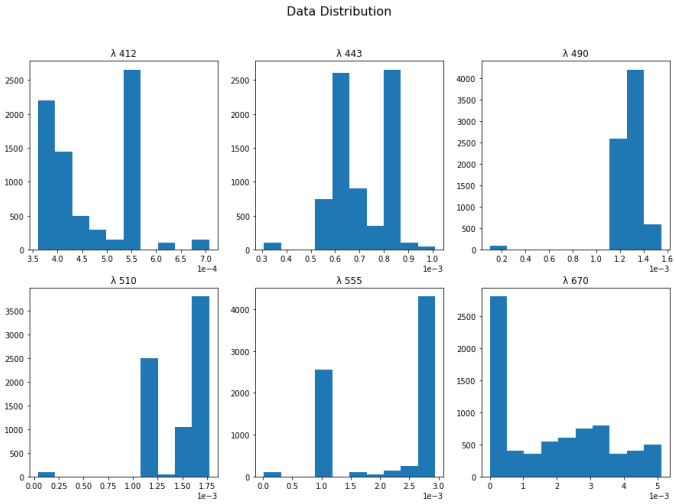
5.2.1 Observed Parameter Scaling

The best observed parameter scaling method varies from problem to problem depending on their distribution. Analyzing the reflectance readings for the various wavelengths (Figure 5.8), it can be seen in 5.8a, that the range in which the data is disposed of is very small (10^{-3}), which means that the model would have difficulty in differentiating close points. To decide what scaling method to utilize, a deeper study of the data distribution is required.

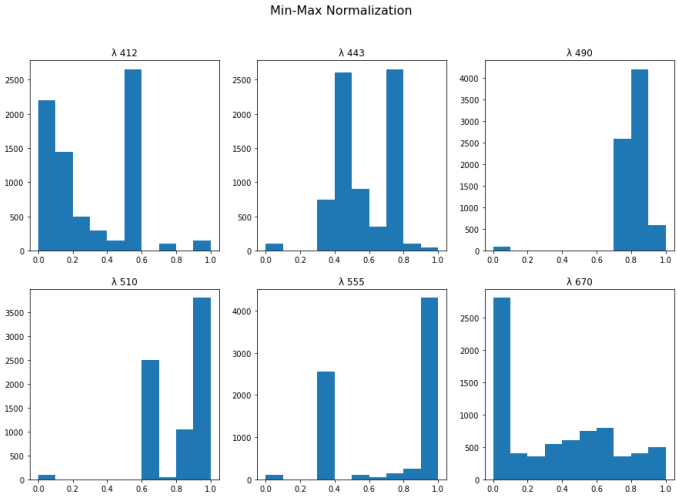
As presented in 4.1.2, the main problem of Min-Max normalization is that outliers tend to "clump up" the data. As in this problem the observed parameters do not follow a uniform distribution, this effect can be seen to great extend in 5.8b where for λ values of 412,490 and 510, >90% of the data is restrained to 60% or less of the total range $[0,1]$.

This problem is mitigated when using Standardization 5.8c, which allows for a more spread-out data range and outliers do not have as much impact on the scale.

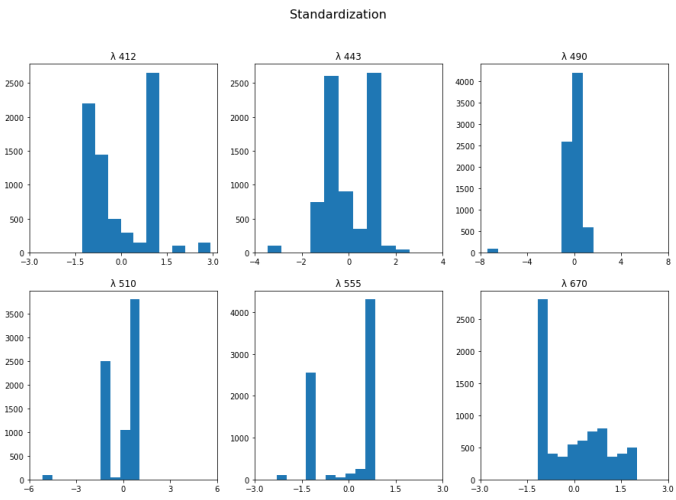
Having this two factors in consideration, standardizing the observed parameters will provide a more optimal training which translates to a higher model accuracy.



(a) No Scaling



(b) Min-Max Normalization



(c) Standardization

Figure 5.8: Scaled reflectance readings for the different wavelengths (7500 points)

5.2.2 Impact of Sampling Proportions during Training

In this subsection, we aim to evaluate the effects that the amount of sampled data from the error density distribution has on the training process. Using a model composed of 3 hidden layers with 64 neurons each, we evaluated the training performance for 10 different sampling percentages (0%,5%,10%,20%,25%,33%,50%,66%,75% and 100%) over batches of 32 points with epochs of 10^4 data points.

The sampling will start with fully random batches for the first epochs and gradually increase the amount of points up to the evaluated amount following this equation:

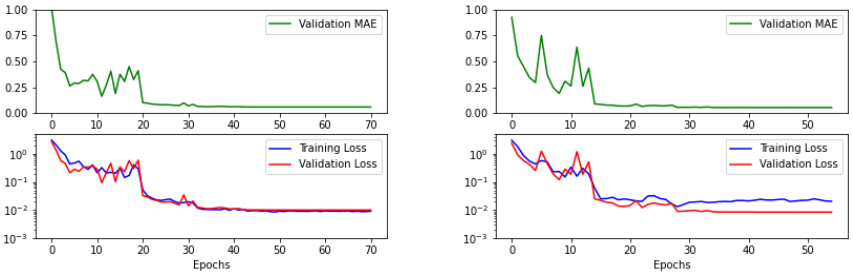
$$\frac{e^x - e^{-x}}{e^x - e^{-x}} \times \frac{percentage}{100} \times batch\_size \qquad \text{where} \quad x = \frac{\#current\_epoch}{10}$$

The evaluated parameters are judged based on model accuracy and converging time.

Percentage	0%	5%	10%	20%	25%	33%	50%	66%	75%	100%
Batch Points	0	1	3	6	8	11	16	21	24	32
Loss (MSE)	0.0103	0.0089	0.0078	0.0104	0.0094	0.0098	0.0183	0.0235	0.0263	0.0341

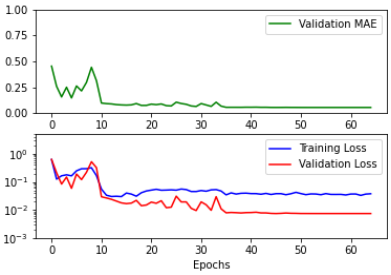
Table 5.4: testing loss with different percentage sampling from error distribution

As can be seen from Table 5.4 and Figure 5.9, sampling up to 33% of the batch size from the error distribution allows for a greater network accuracy and a faster convergence. This happens because by generating data points where the error is higher allows the model to better learn to that domain. However, when overdoing it the model tends to overfit the harder data, learning transformations that do not extend to the remaining domain, which jeopardizes training. From these results we believe that sampling 10% from the error distribution and 90% randomly is capable of maintaining the dataset stability while also giving emphasis during training to the points where the network has the lowest accuracy providing a faster convergence while maintaining or even increasing training accuracy.

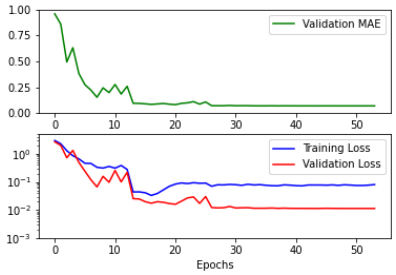


(a) 0%

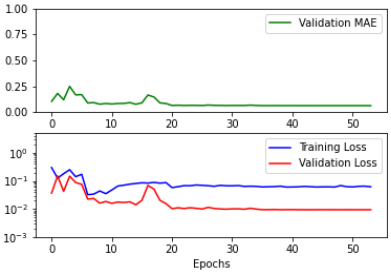
(b) 5%



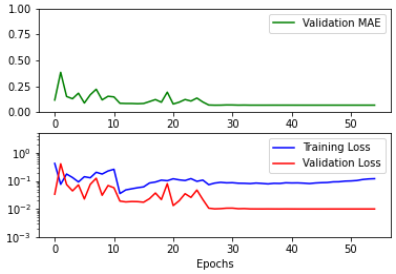
(c) 10%



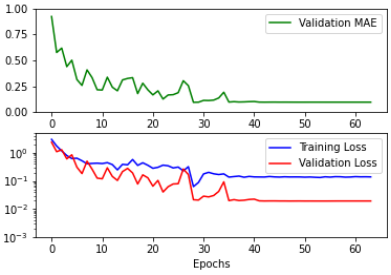
(d) 20%



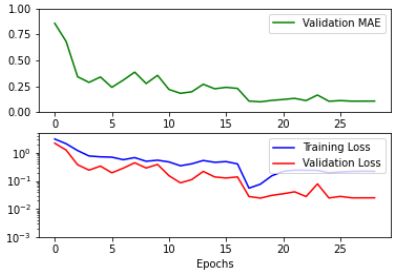
(e) 25%



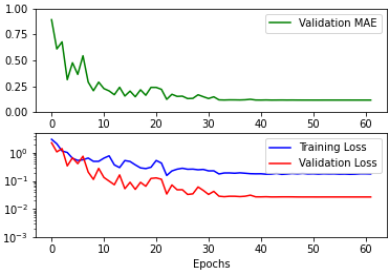
(f) 33%



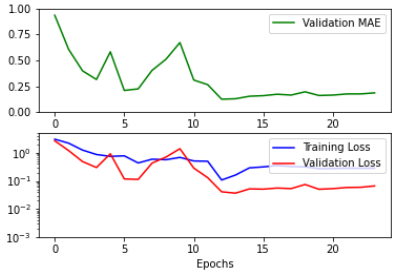
(g) 50%



(h) 66%



(i) 75%



(j) 100%

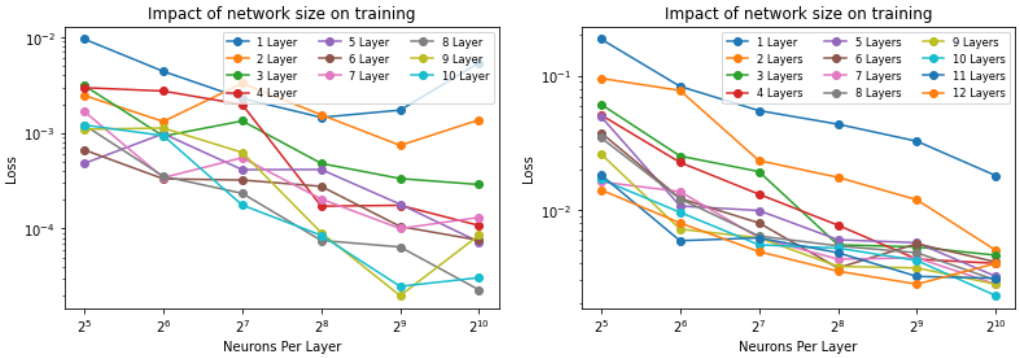
Figure 5.9: Training history with different percentage sampling from error distribution

5.2.3 Model Development

The used model also depends on the problem at hand, this subsection, illustrates how a model can be developed in order to acquire the best accuracy. Starting with a simple model and increase its complexity until the desired results are attained or a plateau is reached, this methodology was used to develop the models for the two studied problems.

As explained in Section 4.2.1, the number of neurons in the input and output layer depend entirely on the number of dependent and independent variables. As the forward model for the Projectile Motion function is a $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ our regression model will have 2 neurons in the input layer and 2 neurons on the output layer. Differently being the forward model of the Ocean Color function a $\mathbb{R}^3 \rightarrow \mathbb{R}^6$ its inverse will be a $\mathbb{R}^6 \rightarrow \mathbb{R}^3$ function, therefore, our regression model will have 6 neurons in the input layer and 3 neurons on the output layer.

As neither problem follows a linear equation it is required to have at least one hidden layer in order to perform non-linear transformations. The chosen methodology to develop the best possible model consisted in starting with a single hidden layer of 32 neurons and iteratively increase the amount of neurons by a factor of 2. Once reached 2^{10} neurons, an additional layer is its added to the model and the number of neurons per layer is reset to 32. This process repeats until a plateau is reached. Figure 5.10 shows the impact of this methodology during training.



(a) Projectile Motion (b) Ocean Color

Figure 5.10: Impact of network size

As can be seen, the best attained model for Projectile Motion and Ocean Color possessed 9 hidden layers of 512 neurons and 10 hidden layers with 1024 neurons respectively. With these results we can confirm that the ocean color problem has a higher complexity compared to the projectile motion problem as similar networks possess significantly different losses. These networks dimensions are unexpectedly large, usually large networks tend to overfit the data and provide poor results compared to simpler ones, however this did not prove to be true with our methodology as larger networks have a lower loss value. A possible explanation for this occurrence is that our adaptive data generator prevents overfitting with the insertion of more data. As previously explained,

the particularity of this methodology is that we possess a seemingly endless dataset which paired with the adaptiveness of the generator allows for the model to correct the domain that were prejudiced during training due to overfitting, which allows for larger networks to thrive.

However it is to have in consideration that this larger networks did not provide that large of an improvement, as for example the Ocean Color model, Figure 5.10b, a model with 2 hidden layers with 1024 neurons each, attained a similar loss value (notice the logarithmic scale) compared to the more complex model.

For this reason, we suggest that the user picks an accuracy threshold, which depends on the problem (e.g. 1% or 10^{-2} if the data is normalized) and use the simplest network that allows it. With this approach, the dimension of the model only depends on the problem's complexity and the users accuracy requirements. This relation between problem and the model complexity can be seen in great extent in Figure 5.10, where if set a loss value threshold of 10^{-2}, for the Ocean Color problem we would require a model with 5 layers of 128 neurons while for the Projectile Motion problem a simple model with a single layer of 32 neurons is enough.

5.2.4 Impact of Measurements Uncertainty

An important contribution of this approach to the OC community is its ability to address different assumptions on the accuracy of the measurements to understand how the satellite's inherent measurements uncertainty impacts the predictions of the retrieved OC compounds. To analyse the impact of these uncertainties on model predictions we will resort to test #2 from the Probabilistic Constraint Approach [19] presented in Table 3.3.

The test case #2 has as observed parameters Chla, CDOM, and NPPM equal to 5.000, 0.035 and 1.141 respectively. These values applied to the forward model result in simulated observations of $[5.469 \quad 5.831 \quad 7.611 \quad 7.746 \quad 7.774 \quad 1.811] \times 10^{-3}$. If directly applied to the best developed model, the predicted measurements of Chla, CDOM, and NPPM are 4.9628, 0.0348, and 1.1381, which corresponds to observations of $[5.471 \quad 5.833 \quad 7.611 \quad 7.743 \quad 7.766 \quad 1.81] \times 10^{-3}$.

Assuming a Gaussian error distribution with mean $\mu_i = 0$ and a standard deviation σ_i equal to 5% of the obtained measurement (except for λ_6 where it is 6%). The impact of the measurement inaccuracy can be seen in Tab. 5.5.

In Fig. 5.11, Fig. 5.12 and Fig. 5.13 are presented the probability density distribution of test #2 computed using the Monte Carlo method with 10^5 points. Fig 5.11 illustrate the obtained joint distributions allowing us to identify regions of maximum likelihood (purple represents less likely regions and yellow is the more likely). Figure 5.12 shows the marginal distributions for the junction of the different compounds and Fig. 5.13 the marginal distribution for each compound. For this test, we attained a mean value for the compounds of 5.401 X 0.007 X 1.112 and a standard deviation of 2.909 X 0.172 X

0.225. From these results, it is clear that uncertainty paired with the model inherent error plays a major role in predictions, especially when working with non-linear problems that due to their instability conditions, a small deviation on the observed parameter may lead to major changes in the predictions. This effect can be seen in Tab. 5.6, where we evaluated different measurement inaccuracies, being the first row computed with the error standard deviation specified in 3.2 and the remaining rows a percentage of that value. With improved measurement accuracy, the expected value converges to the exact value used to simulate the observations and the standard deviation approaches zero. This provides insight on the magnitude of the incurred errors, with different sensor accuracies.

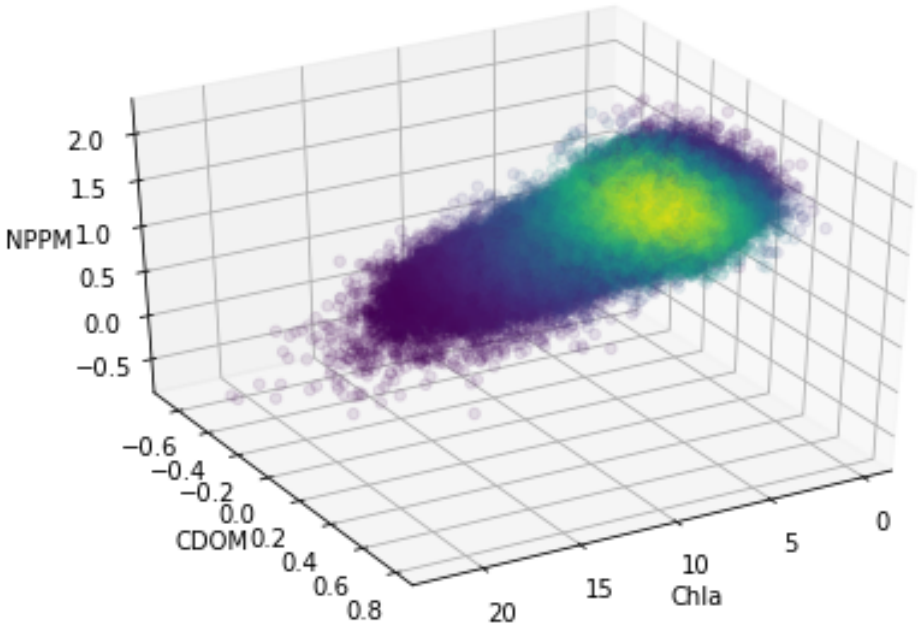
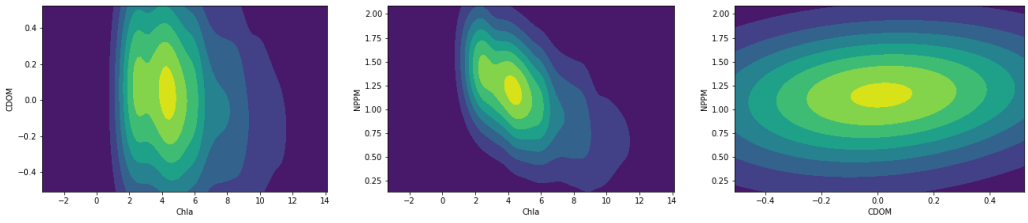
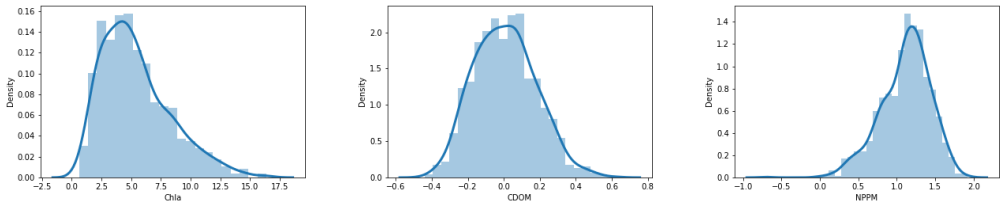


Figure 5.11: Joint distribution computed with Monte Carlo method



(a) Chla and CDOM (b) Chla and NPPM (c) CDOM and NPPM

Figure 5.12: Marginal distributions for the joint combination of two compounds



(a) Chla (b) CDOM (c) NPPM

Figure 5.13: Marginal Distributions for each compound

Test	Chla			CDOM			NPPM			MSE
	Model Pred	μ	σ	Model Pred	μ	σ	Model Pred	μ	σ	
1	0.930	0.988	0.526	0.017	0.0172	0.090	0.291	0.290	0.158	0,0001
2	4.9628	5.401	2.909	0.034	0.007	0.172	1.138	1.112	0.225	0,0541
3	10.076	11.679	8.198	1.865	1.854	0.292	0.050	-0.192	1.064	0,9601
4	-0.759	-0.773	0.260	-0.063	0.117	0.084	0.485	0.496	0.176	0,2037
5	-0.605	-0.692	0.531	0.092	0.067	0.138	0.963	0.993	0.214	0,1611
6	-0.501	-0.459	1.259	-0.004	0.042	0.185	5.041	5.074	0.475	0,0726
7	-0.282	-0.263	0.217	-0.013	0.008	0.054	0.431	0.445	0.126	0,0269
8	0.009	0.043	0.618	0.489	0.505	0.118	0.497	0.458	0.121	0,0012
9	0.028	0.051	0.532	2.040	2.104	0.259	0.475	0.438	0.205	0,0058
10	0.078	0.092	0.240	0.281	0.288	0.061	0.123	0.125	0.098	0,0044
11	1.042	1.194	1.203	4.990	4.923	0.619	0.500	0.517	0.231	0,0146
12	9.963	10.697	6.438	0.517	0.514	0.173	2.005	1.914	0.554	0,1645

Table 5.5: Results for the 12 experimental cases extracted from [19] applying the Machine Learning Approach

	Chla		CDOM		NPPM	
	μ	σ	μ	σ	μ	σ
100%	5.401	2.909	0.007	0.172	1.112	0.225
50%	5.081	1.523	0.021	0.099	1.129	0.168
10%	4.966	0.297	0.033	0.021	1.138	0.030
5%	4.967	0.149	0.034	0.011	1.138	0.015
1%	4.962	0.029	0.035	0.002	1.138	0.003

Table 5.6: Mean and standard deviation values obtained for different accuracies

5.2.5 Comparison of Methodologies

To validate our approach we will compare it to the work in [19], we have access to their obtained results on 12 simulated experiments representative of the different seawater types that can be found in nature (Table 5.8). The comparison will have into consideration two main topics: execution time and prediction accuracy.

Test	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
Execution Time	110	119	121	87	113	92	72	73	71	78	69	89

Table 5.7: Execution time in seconds for the Probabilistic Constraint Approach

The first main advantage of the Machine Learning Approach this thesis proposes is on execution time. Unlike the Probabilistic Constraint Approach, where for each new measurement its necessary to recompute the expected prediction through the complex set of constraints, this approach to creates a continuous function that approximates the inverse function through the employment of neural networks, this proportionate the capability of making predictions in a negligible time once the model is trained. The Probabilistic Approach took 18 minutes to predict the 12 tests (Table 5.7), that compared to the 12 minutes it took to train the best model used that is now able to make endless predictions in a matter of instants as the act of predicting is the application of successive of multiplications and sums that take negligible time to compute.

When it comes to model accuracy, over the 12 tests, the Machine Learning Approach attained competitive results compared to the Probabilistic Constraint Approach (Table 5.8). With the proposed methodology we were able to ensure that the true value is within the minimum ($Q1 - 1.5 \times IQR$) and maximum values ($Q3 + 1.5 \times IQR$), even though this range is recurrently larger for the ML Approach compared to the CP Approach, the latest does not guarantee it, as can be seen in test #3.

For the tests with values closer to the mid-point of the user defined boundaries, the model is the most accurate, where the median value is a close approximation to the true value and its guaranteed that the latest is inside the Interquartile Range, this is between the 25th to the 75th percentile.

However, this methodology did have poor results for tests that involved predicting the values of model parameters close to the lower bound of the user defined boundaries (tests #4 to #7). The model had the most difficulty predicting the values of Chla, especially for values close to 0. This compound had a special particularity compared to the other two as even though unfeasible in real life, the function is defined for negative values of CDOM and NPPM, unlike the Chla whose impact on the direct model is inside a square root making it undefined for values lower than 0. This turning point where the function becomes ambiguous or not defined (Chla < 0, CDOM < 0 and NPPM < 0) makes it harder for the model to learn without jeopardizing the remaining domain's accuracy.

Overall, this methodology proved to be a competitor to the already existing ones. The low prediction time paired with its high accuracy makes this a framework that is suitable

not only for applications that require near instantaneous responses but is also able to work with larger dataset in a timely way, which the previous methodologies were not able to accomplish. This does not mean that it is superior in every single way, as for values that are close to the ambiguity turning point, the model may predict unfeasible values. For this values, we believe that pairing this approach with the Probability Constraint Approach will provide added benefits on accuracy despite requiring additional processing.

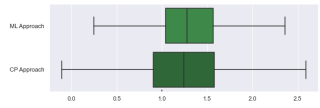
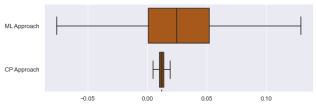
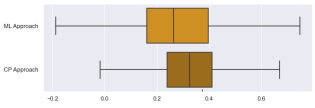
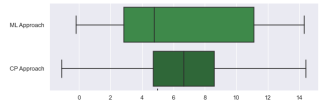
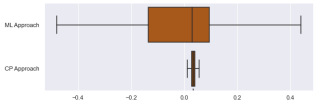
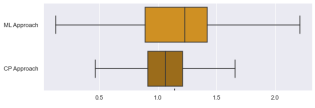
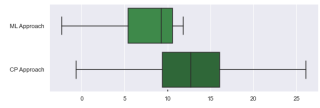
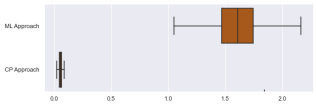
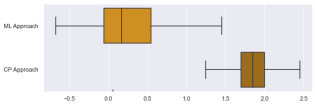
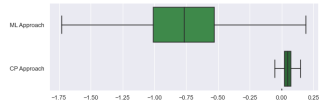
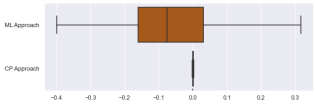
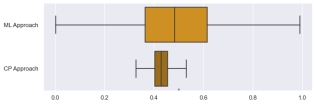
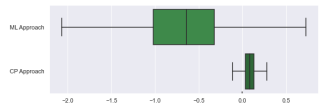
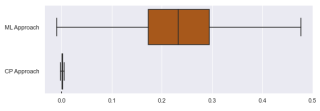
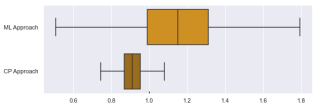
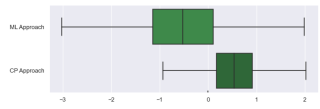
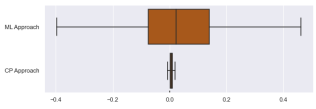
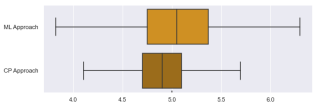
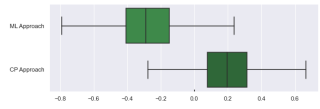
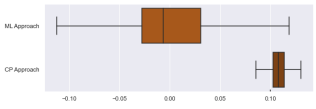
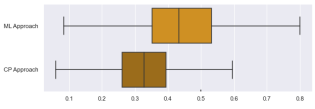
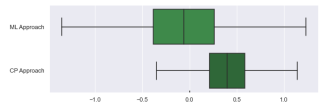
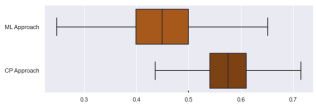
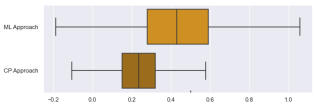
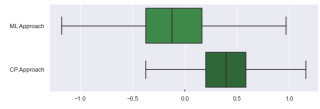
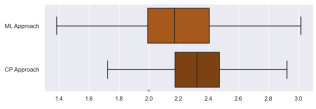
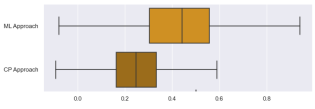
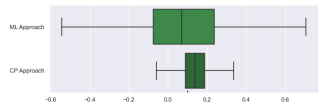
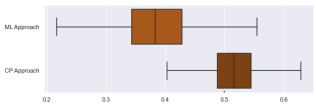
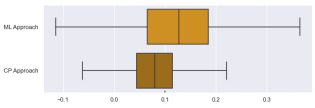
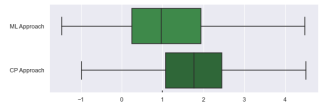
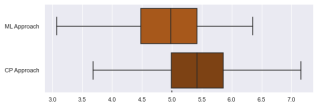
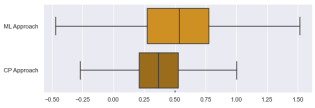
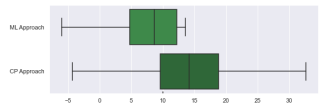
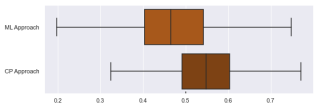
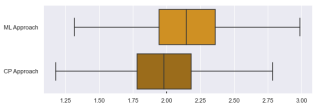
Test	Chla	CDOM	NPPM
#1			
#2			
#3			
#4			
#5			
#6			
#7			
#8			
#9			
#10			
#11			
#12			

Table 5.8: Comparison of results from the Probabilistic Constraint Approach (PC) and Machine Learning Approach (ML) for the 12 tests in [19]

6

Conclusion

With this dissertation, we proposed to evaluate the performance of applying deep learning techniques to support reasoning for nonlinear inverse problems.

The central questions for our research were as follows:

1. How to map from the observable parameters to the model parameters using neural networks?

2. How to generate data through the employment of the forward model to train the network?

3. How can uncertainty of errors on the observable data be represented and reproduced to train the network?

4. What is the efficiency of this method compared to previously developed ones?

All inverse problems have an inherent complexity that varies for different problems therefore there is no unique solution when it comes to mapping the inverse model. Throughout this dissertation, we evaluated different regression methods in order to solve the task at hand. One studied hypothesis was the employment of the forward model to measure the error on the model predictions, with this methodology we expected to have a faster and smoother convergence to a minimum however this did not prove to be true as already existing loss functions such as MSE wielded better results. Another parameter of the study was which procedure to use in order to input data into the model, we reached inconclusive results on which practice is preferable as the distribution of the observed parameters depends on the forward model that the user is trying to compute. When it comes to the model output format, we concluded that an unbounded function such as the linear output function had the best performance when paired with an unscaled output format defined by the user. Having these results into consideration, we believe that a linear output network with standardized observed parameters and unscaled model parameters can result in a highly efficient regression model, but nevertheless, if the user aims to achieve maximum accuracy a study of the data distribution and model adjustment has to be made.

One major complexity of our approach was the nonexistence of a dataset, instead, we had a generation method in the form of a forward model. In this dissertation, we proposed the employment of this forward function to generate a dataset that would train the regression model. Our initial approach consisted in creating a large dataset and using it to train the model, however, this naive technique did not ensure proper training resulting in the model having low accuracy and tending to overfit. We settled on an adaptive data generator, inspired by the GTN architecture from [17], that adjusts the computed data into the domain where the network has the lowest accuracy/ highest error. This technique allowed us to train the model to its maximum potential while minimizing overfitting on a single dataset, decreasing training time, and maximizing the model's accuracy.

With our work, we aimed to allow the user the ability to address different assumptions on the accuracy of the measurements to understand how these may affect the uncertainty of the measurements. This ability would be very important for the OC community as this study would allow to define accuracy requirements for the radiometric sensors to guarantee specified levels of uncertainty for the estimated concentrations. Here we assumed that the error follows a Gaussian distribution with a mean of 0 and a standard deviation equal to a defined percentage of the total measurement. With this distribution, we sample multiple possible observations that we then feed to the model for predictions. With these multiple predictions, we were able to compute the probability distribution and provide the most probable value for the given measurement.

Overall, the approach proposed by this thesis proved to very viable. This approach is faster compared to the Probability Constraint Approach as once the model is trained it allows for almost instantaneous predictions with elevated accuracy, which the latest is not able to perform, because for each new observation the latest necessitates to re-propagate the constraints to compute the new prediction, which is very time consuming. However, for predictions that involve values of the model parameters very close to the turning point where the direct model becomes ambiguous or not defined, the Machine Learning Approach had worse performance, with prediction values outside the feasible and defined domain. For this values, we believe that applying the Probability Constraint Approach might provide added benefits on accuracy despite requiring additional computational time.

Bibliography

[1] Z. Pizlo. "Perception viewed as an inverse problem". In: *Vision Research* 41.24 (2001), pp. 3145–3161. ISSN: 0042-6989. DOI: 10.1016/S0042-6989(01)00173-0. URL: https://www.sciencedirect.com/science/article/pii/S0042698901001730 (cit. on p. 1).

[2] A. Tarantola. *Inverse Problem Theory and Methods for Model Parameter Estimation*. Society for Industrial and Applied Mathematics Philadelphia, 2005 (cit. on pp. 1, 22, 26).

[3] K. R. Heidi M. Dierssen. "Remote Sensing of Ocean Color". In: (2013). DOI: 10.1007/978-1-4614-5684-1_18 (cit. on p. 3).

[4] I. S. Robinson. *Discovering the Ocean from Space*. Springer, 2010 (cit. on p. 3).

[5] L. P. Andre Morel. "Analysis of variation in ocean colour". In: *Limnology and Oceanography* (1977). DOI: 10.4319/lo.1977.22.4.0709 (cit. on pp. 3, 32).

[6] E. Team. *What is Machine Learning? A Definition*. 2020. URL: https://www.expert.ai/blog/machine-learning-definition/ (cit. on p. 5).

[7] L. Krippahl. *Aprendizagem Automática (Machine Learning) - Lecture Notes*. 2018 (cit. on pp. 5, 14, 17, 18).

[8] L. Krippahl. *Aprendizagem Profunda - Lecture Notes*. 2021 (cit. on pp. 6, 14, 17, 18).

[9] Y.-S. Park and S. Lek. "Artificial Neural Networks". In: *Developments in Environmental Modelling* (2016), pp. 123–140. DOI: 10.1016/b978-0-444-63623-2.00007-4 (cit. on pp. 6, 8).

[10] K. Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(91)90009-T. URL: https://www.sciencedirect.com/science/article/pii/089360809190009T (cit. on p. 6).

[11] J. Brownlee. *A gentle introduction to the rectified linear unit (ReLU)*. Aug. 2020. URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ (cit. on p. 9).

[12] G. Mayanglambam. *loss vs no. of the epoch*. [Online; accessed September 4, 2022]. 2020. URL: https://miro.medium.com/max/1400/1*R12QVNFn-46IPewAMxde2w.png (cit. on p. 12).

[13] S. Bock, J. Goppold, and M. Weiß. "An improvement of the convergence proof of the ADAM-Optimizer". In: Apr. 2018 (cit. on p. 14).

[14] J. Adler and O. Öktem. "Solving ill-posed inverse problems using iterative deep neural networks". In: *Inverse Problems* 33.12 (2017), p. 124007. DOI: 10.1088/1361-6420/aa9581 (cit. on pp. 14, 27, 29).

[15] J. L. Mueller and S. Siltanen. *Linear and Nonlinear Inverse Problems with Practical Applications*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2012. DOI: 10.1137/1.9781611972344. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611972344. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611972344 (cit. on pp. 18, 23).

[16] J. Brownlee. *Smote for imbalanced classification with python*. Mar. 2021. URL: https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/ (cit. on p. 19).

[17] F. P. Such et al. "Generative Teaching Networks: Accelerating Neural Architecture Search by Learning to Generate Synthetic Training Data". In: *CoRR* abs/1912.07768 (2019). arXiv: 1912.07768. URL: http://arxiv.org/abs/1912.07768 (cit. on pp. 20, 71).

[18] J. Hadamard. "Lectures on Cauchy's Problem in Linear Partial Differential Equations". In: (1923) (cit. on p. 23).

[19] E. Carvalho, J. Cruz, and P. Barahona. "Probabilistic constraints for nonlinear inverse problems An ocean color remote sensing example". In: (2013). DOI: 10.1007/s10601-012-9139-6 (cit. on pp. 24, 26, 27, 33, 34, 59, 64, 66, 67, 69).

[20] L. Granvilliers, J. Cruz, and P. Barahona. "Parameter estimation using interval computations". In: *SIAM Journal on Scientific Computing* 26.2 (2004), pp. 591–612. DOI: 10.1137/s1064827503426851 (cit. on p. 25).

[21] A. Goldsztejn, J. Cruz, and E. Carvalho. "Convergence analysis and adaptive strategy for the certified quadrature over a set defined by inequalities". In: *Journal of Computational and Applied Mathematics* (Apr. 2014), pp. 543–560. URL: https://hal.archives-ouvertes.fr/hal-00911289 (cit. on p. 26).

[22] H. Li et al. "Nett: Solving inverse problems with deep neural networks". In: *Inverse Problems* 36.6 (2020), p. 065005. DOI: 10.1088/1361-6420/ab6d57 (cit. on pp. 27, 29).

[23] B. Kelly, T. P. Matthews, and M. A. Anastasio. "Deep Learning-Guided Image Reconstruction from Incomplete Data". In: *CoRR* abs/1709.00584 (2017). arXiv: 1709.00584. URL: http://arxiv.org/abs/1709.00584 (cit. on pp. 27, 29).

[24] E. Celledoni et al. "Equivariant neural networks for inverse problems". In: *Inverse Problems* 37.8 (2021), p. 085006. DOI: 10.1088/1361-6420/ac104f (cit. on pp. 27, 29).

[25] L. Ardizzone et al. "Analyzing Inverse Problems with Invertible Neural Networks". In: *CoRR* abs/1808.04730 (2018). arXiv: 1808.04730. URL: http://arxiv.org/abs/1808.04730 (cit. on pp. 27, 31).

[26] J. Kruse et al. "Benchmarking Invertible Architectures on Inverse Problems". In: *CoRR* abs/2101.10763 (2021). arXiv: 2101.10763. URL: https://arxiv.org/abs/2101.10763 (cit. on pp. 27, 31).

[27] S. Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (cit. on p. 27).

[28] R. Prabhu. *Understanding of Convolutional Neural Network (CNN) — Deep Learning*. 2018. URL: https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148 (cit. on p. 27).

[29] J. Brownlee. *How Do Convolutional Layers Work in Deep Learning Neural Networks?* 2020. URL: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/ (cit. on p. 28).

[30] L. Dinh, J. Sohl-Dickstein, and S. Bengio. "Density estimation using Real NVP". In: *CoRR* abs/1605.08803 (2016). arXiv: 1605.08803. URL: http://arxiv.org/abs/1605.08803 (cit. on p. 29).

[31] S. Groom et al. "Satellite Ocean Colour: Current Status and Future Perspective". In: *Frontiers in Marine Science* 6 (2019). DOI: 10.3389/fmars.2019.00485 (cit. on p. 32).

[32] C. R. McClain, G. C. Feldman, and S. B. Hooker. "An overview of the seawifs project and strategies for producing a climate research quality global ocean bio-optical time series". In: *Deep Sea Research Part II: Topical Studies in Oceanography* 51.1-3 (2004), pp. 5–42. DOI: 10.1016/j.dsr2.2003.11.001 (cit. on p. 33).

[33] M. S. Twardowski et al. "Optical backscattering properties of the "Clearest"natural waters". In: *Biogeosciences* 4.6 (2007), pp. 1041–1058. DOI: 10.5194/bg-4-1041-2007 (cit. on p. 33).

[34] C. R. McClain, G. Meister, and Ioccg. *Mission requirements for future ocean-colour sensors*. 2012. URL: http://dx.doi.org/10.25607/OBP-104 (cit. on p. 33).

[35] Z. G and V. K. *Field Radiometry and Ocean Color Remote Sensing*. Dordrecht (The Netherlands): Springer, 2010, pp. 307–334. ISBN: 978-90-481-8680-8. DOI: 10.1007/978-90-481-8681-5_18 (cit. on p. 33).

[36] M. Shanker, M. Hu, and M. Hung. "Effect of data standardization on neural network training". In: *Omega* 24.4 (1996), pp. 385–397. ISSN: 0305-0483. DOI: https://doi.org/10.1016/0305-0483(96)00010-2. URL: https://www.sciencedirect.com/science/article/pii/0305048396000102 (cit. on p. 36).

[37] K. He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. DOI: 10.48550/ARXIV.1502.01852. URL: https://arxiv.org/abs/1502.01852 (cit. on p. 40).

[38] S. C. Instructors. *CS231n: Deep Learning for Computer Vision*. URL: https://cs231n.github.io/neural-networks-3/#hyper (cit. on p. 43).

I

Annex 1

I.1 Translate the Function

Initially the user provides a string that represents the function $f : M \rightarrow O$, that its aimed to be computed the inverse of. With $M \subseteq \mathbb{R}^n$ and $O \subseteq \mathbb{R}^k$, where n and k respectively the number of model and observed parameters.

This string is then parsed into two different function. The first one is used to generate the different points that will compose the dataset and the second function, is to be used in order to evaluate the performance of using the forward model when training the network.

I.1.1 Lexer

A tokenizer splits the string into individual tokens. For example, if the user passes the function: $f(x) = 2 * 3 + x$, the tokenizer splits it into the tokens '2','*','3','+' and 'x'.

In this thesis we resorted to lex.py, the PLY lexer module. This module allows to break input text into a collection of tokens specified by a collection of regular expression rules like previously explained. For example, the previous case would be: ('NUMBER',2), ('TIMES','*'), ('NUMBER',3), ('PLUS','+'), ('NAME','x').

Tokens are usually given names to indicate what they are and their respective regular expression rule (Table I.1).

Token	Regular Expression
NAME	r'[a-zA-Z_][a-zA-Z0-9_]*'
NUMBER	r'\d+(\.\d+)?'
PLUS	r'\+'
MINUS	r'-'
TIMES	r'*'
DIVIDE	r'/'
POWER	r'\^'
LPAREN	r'\('
RPAREN	r'\)'

Table I.1: Tokens and their regular expressions

The module provides an external interface in the form of a function that returns the next valid token on the input stream that is used by the parser to retrieve tokens and invoke grammar rules.

I.1.2 Parsing the Forward Model

A parser is used to recognize language syntax that has been specified in the form of a context free grammar. In this thesis the parser used is yacc.py, the parsing module from PLY.

The syntax is usually specified in terms of a Backus–Naur form grammar. The grammar developed, for this thesis, to parse arithmetic, trigonometric and polynomial expressions is as shown in Table I.2.

expression:	expression PLUS expression
	| expression MINUS expression
	| expression TIMES expression
	| expression DIVIDE expression
	| expression POWER expression
	| MINUS expression
	| LPAREN expression RPAREN
	| NAME LPAREN expression RPAREN
	| NAME
	| NUMBER

Table I.2: Grammar

As this grammar is ambiguous, yacc.py allows individual tokens to be assigned a precedence level and associativity.

```
precedence = (('left','PLUS','MINUS'), ('left','TIMES','DIVIDE'),
('left', 'POWER'), ('right','UMINUS'))
```

This specifies that PLUS and MINUS have the same precedence level and are left-associative while UMINUS is right-associative. Within the precedence declaration, tokens are ordered from lowest to highest precedence, therefore, TIMES and DIVIDE have higher precedence than PLUS and MINUS since they appear later in the precedence specification.

Once the grammar is defined its required to define the grammar rule functions I.3. As with this function we aim to create large datasets, we resorted to the NumPy library as it provides support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

I.1.3 Parsing the TensorFlow version of the Forward Model

There are two main reasons that made mandatory the parsing of the forward model as a TensorFlow function. When dealing with neural networks libraries like Keras and similar ones, the main object that is manipulated and passed around is the a Tensor.

Expression	Action
expression PLUS expression	$e_1 + e_2$
expression MINUS expression	$e_1 - e_2$
expression TIMES expression	$e_1 \times e_2$
expression DIVIDE expression	e_1 / e_2
expression POWER expression	$e_1^{e_2}$
MINUS expression	$-e_1$
LPAREN expression RPAREN	(e_1)
NAME LPAREN expression RPAREN	executes the NumPy function of NAME on e
NAME	creates a NumPy array with NAME
NUMBER	converts the string NUMBER to a float value

Table I.3: Rule function for each expression using NumPy

Expression	Action
expression PLUS expression	TensorFlow.math.add(e_1, e_2)
expression MINUS expression	TensorFlow.math.subtract(e_1, e_2)
expression TIMES expression	TensorFlow.math.multiply(e_1, e_2)
expression DIVIDE expression	TensorFlow.math.divide(e_1, e_2)
expression POWER expression	TensorFlow.math.pow(e_1, e_2)
MINUS expression	TensorFlow.math.negative(e_1)
LPAREN expression RPAREN	prioritizes e_1
NAME LPAREN expression RPAREN	executes the TensorFlow function of NAME on e
NAME	creates a TensorFlow Variable with NAME
NUMBER	creates a TensorFlow Constant with NUMBER

Table I.4: Rule function for each expression using TensorFlow

A Tensor is the TensorFlow representation of a multidimensional array. Even though there are function that are able to transform Tensors into regular arrays and vice-versa, the main reason that made obligatory this parsing is to compute the derivative of the function in a easier and faster way.

As explained in 2.1.1, in order to apply the gradient descent formula, its required to compute the derivative of the error in regard to the observations. TensorFlow is able to compute this faster, as it is able to be executed in parallel by a GPU, and provide the gradients used to update the weights. So similarly to the NumPy functions, a set of grammar rule function using TensorFlow functions was developed to automatically compute and compile the Tensor version of the forward model (Table I.4).