Operator Learning for Loss Parameter Estimation in Dredging Operations

Marius Kielhöfer

Operator Learning for Loss Parameter Estimation in Dredging Operations

To optimize the suction production on Trailing Suction Hopper Dredgers.

Marius Kielhöfer

A thesis presented for the degree of
Master of Applied Mathematics, to be defended publicly on Thursday October 23, 2025.

Student number:	5125502
Project duration:	September 30, 2024 — October 23, 2025
Thesis committee:	A. Heinlein, TU Delft, Supervisor
	G. Jongbloed, TU Delft
	M. van Gijzen, TU Delft

Operator Learning for Loss Parameter Estimation in Dredging Operations

Marius Kielhöfer

Abstract

Accurate modeling of vacuum dynamics in Trailing Suction Hopper Dredgers (TSHDs) is critical for optimizing suction production and mitigating sensor anomalies. This study proposes a data-driven, physics-guided operator learning framework to estimate the vacuum pressure loss parameter θ, a variable derived from physical principles in dredging operations. Leveraging a modified Deep Operator Network (DeepONet), we introduce attention-based interactions between branches and the trunk network to capture complex dependencies in the sensor data. A local trunk mechanism is introduced to preserve temporal locality across dredging trips.

Due to the nature of a lagging density sensor, we integrate a real-time rolling mean error correction mechanism. This addresses training biases for refined predictions, as well as offering an anomaly detection mechanism. The model is trained and validated on real-world vessel data, including synthetic simulations of vacuum processes, and evaluated using trip-wise and global metrics. Experimental results show that the proposed architecture significantly outperforms the rolling mean baseline setups and the classical DeepONet across accuracy metrics such as the root mean square error (RMSE).

This work demonstrates the value of combining domain knowledge with operator learning techniques in maritime engineering. The proposed framework offers a scalable framework, allowing application across entire fleets for real-time suction production estimation and anomaly detection, contributing to efficient dredging operations.

Foreword

This project has taken me a little over a year to complete, longer than anything else I have consistently worked on. There were many bumps on the road and I would not have been able to overcome those bumps without the help of my supervisors around me. First of all I would like to thank my supervisors from Boskalis: Oscar and Willem. Thank you for offering me this project, the weekly meetings, and all the help you provided along the way. From Boskalis I also wish to express my gratitude to the domain experts: Roland Higler, Arno Nobel and Chris van den Berg. I would have been very lost without you! I would also like to extend my thanks to my daily TU Delft supervisor, Alexander. Thank you for guiding me throughout the thesis, giving constant (and a lot of) feedback, and making the project an overall enjoyable experience. Finally, thank you to my girlfriend Emma for always believing in me and encouraging me to believe in myself. I could not have completed this project without the help of everyone involved!

Marius Kielhöfer

Rotterdam, October 2025

Note on artificial intelligence

Portions of the wording and explanations in this thesis were refined with the assistance of artificial intelligence, more specifically, ChatGPT, to improve clarity and readability of the report.

Contents

1 Introduction

As per the Boskalis 2023 sustainability report [5], the dredging and inland infra division accounted for 529,000 metric tons of carbon dioxide emissions in 2023. To combat these high emissions, control systems are being implemented on dredging vessels. These control systems are estimated to save a minimum 5% of emissions per dredging vessel, which could save about 25,000 metric tons of emissions every year. This would not only result in a 5% increase in production because the saved fuel can be used for other purposes, but also make the process more sustainable. The crews on the dredging vessels however, do not yet make much use of the control systems yet.

The control systems operate through sensor readings. Since the sensors are known to be faulty at times, the crews onboard do not trust the exact readings (O. van de Ven, personal communication, November 2024). Boskalis vessels can operate in harsh conditions from storms and strong currents, causing damage and misalignment of the sensors. This in turn causes the unwillingness for the crews to operate the control systems.

The focus of the report will be on the vacuum process and the sensors related to it. The goal of the report is double fold. First, we set up a system ourselves with the aim of increasing production and reducing emissions. The second aim is that by implementing this system, several sensors are checked in the process, thus improving the trustworthiness of the sensors.

The system that we implement consists of estimating a pressure loss parameter: θ. According to dredging literature [57], we know what variables this loss parameter depends on. These variables are all logged in the sensor data. The training and testing datasets were generated by defining relations that simulate the behavior of the loss parameter θ. This way, we avoid having to rely on unreliable sensor data, and can work in a controlled environment.

We set up an operator learning model that maps multiple input variables, such as the mixture velocity, observed over a time window of length w to the loss parameter θ. More formally, given m input variables $(f_1, \ldots, f_m)$ defined

over time steps $(t_{s_k}, \ldots, t_{e_k})$ length w, the operator maps:

$$f_1(t_{s_k}, \ldots t_{e_k}) \ldots f_m(t_{s_k}, \ldots, t_{e_k}) \rightarrow \theta(f_1, \ldots, f_m)(t_{s_k}, \ldots, t_{e_k})$$

This operating learning model is then trained and tested on both simulated and real life noisy data.

Throughout this report, we begin by introducing the dredging process with a focus on relevant onboard sensors, followed by a detailed examination of the vacuum process. A review of related literature on sensor modeling in dredging contexts is then provided in section 3 to motivate the research questions. We then introduce the concept of operator learning and present our proposed architecture: a modified DeepONet designed to capture complex temporal and cross-sensor interactions. This section also discusses the corrector model and sensor outlier detection methodology. In section 5, the structure of the dataset and its integration into the architecture, along with a description of the training procedures are discussed. Finally, the model's performance results are presented and analysed, after which the report concludes.

2 Application Domain

In this section, we explain all the necessary details about dredging. We showcase how a dredging vessel works, which sensors are important, and where they are located. We go into the specifics of the vacuum process and suction production optimization.

2.1 Dredging

Dredging is an engineering process that involves removing and relocating debris and other materials from the bottom of oceans. Dredging is crucial in numerous industrial and environmental applications, including land reclamation and infrastructure development. By clearing and relocating sediment, dredging creates new opportunities for construction and enhances the usability of aquatic environments.

In land reclamation, dredged material is used to create or expand land areas, particularly in coastal and urban regions. This method is widely employed to address the growing demand for space in densely populated areas. In addition to land reclamation, dredging is integral to the construction of harbors and airports, especially those built on or near water. By preparing underwater foundations and ensuring safe waterways, dredging enables the establishment of essential transportation.

2.1.1 Trailing Suction Hopper Dredger

There are different types of vessels associated with dredging, all of which serve different purposes. For this research and the report, we focus on the Trailing Suction Hopper Dredger (TSHD). A TSHD is a specialized type of dredging vessel designed for the removal and transportation of material from the seabed. They are used in practically all dredging operations. The primary advantage of a TSHD is its ability to efficiently collect and transport large volumes of dredged material over long distances.

The working mechanism of a TSHD involves several key components. The dredger is equipped with one or more suction pipes, each fitted with a draghead at the seabed end. As the vessel moves forward, the draghead is lowered onto the seabed and sediment is loosened and "dredged" up through the suction pipe by powerful pumps; see fig. 1. The dredged material is then transported into one or more hoppers: a large storage compartment within the vessel. Once the hoppers are full, the TSHD can transport the dredged material to the designated site. The material can be discharged in various ways, including through bottom doors that open to release the sediment, or by using pumps to offload the material through a pipeline. Some TSHDs are also equipped with rainbow discharge capabilities, allowing them to spray the dredged material over specific areas.

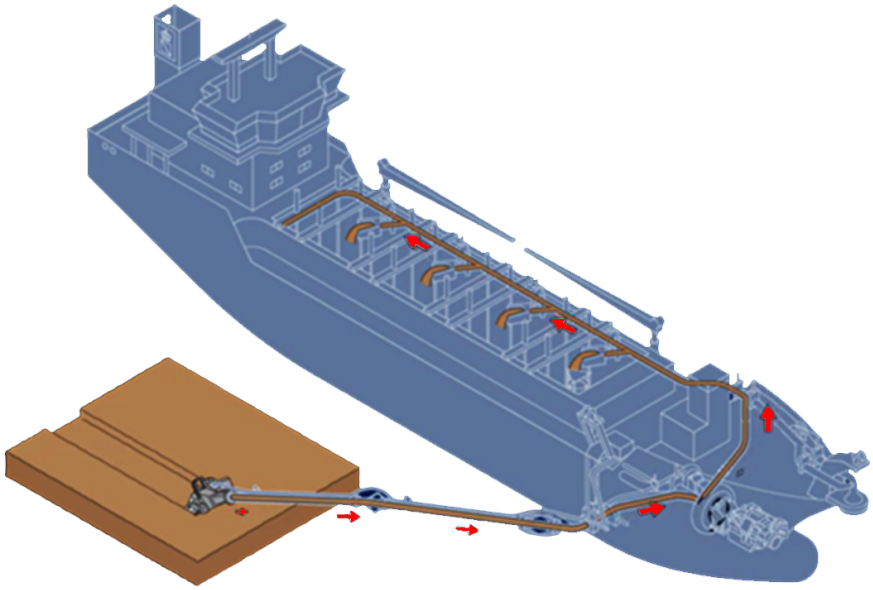


Figure 1: Image of a TSHD dredging sand. The draghead is dragging across the seabed to pick up the sand, this is then dredged up by the dredegepump. The material is stored as a mixture of sand and water in the hoppers of the vessel, ready to be offloaded when all the hoppers are full. Taken from [4].

2.1.2 Sensor deployment overview

This research focuses on specific processes in a dredging ship, and the sensors related to these processes.

During the dredging process, a pump is mounted on the side of the vessel to extract the loosened material. The system operates as a centrifugal pump, as illustrated in fig. 2. Its lower section generates the suction needed to lift the mixture from the seabed, while the upper section transports the dredged material into the hoppers.

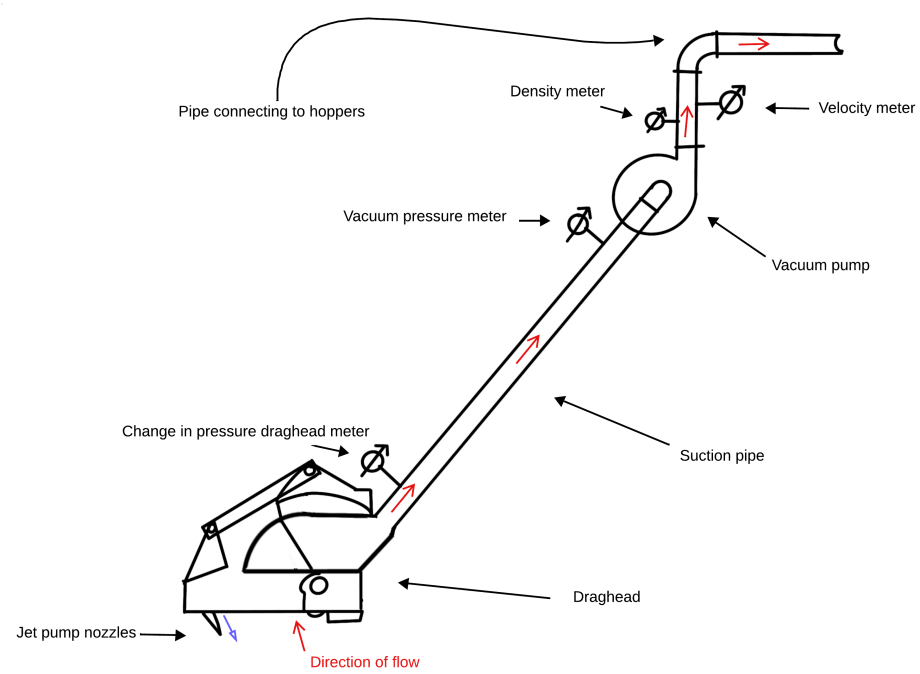


Figure 2: Dredgepump diagram with location of sensors and components. The red arrows represent the flow of the slurry through the dredgepump. The blue arrow shows the directions of the jet pumps, which support the dredgepump.

In front of the vacuum pump is the vacuum sensor. The vacuum sensor measures the pressure in the lower part of the pipe, which dredges up the material. The sensors that keep track of the flow are the mixture velocity meter and the density meter. The mixture velocity sensor is located in the upper part of the

dredgepump. As the name suggests, this sensor measures the velocity of the material that passes through it. Similarly, the mixture density sensor measures the density of the mixture passing through the dredgepump.

The jet pump nozzles on the draghead are used to assist the dredging process by loosening seabed material before it enters the suction mouth. When activated, these high-pressure water jets cut through soil layers, reducing suction resistance and improving the efficiency of material intake. The jet pump pressure is continuously measured and can be manually adjusted, allowing control over the cutting intensity depending on soil conditions.

Some other important sensors include the draught and hopper sensors. The draught sensor checks the draught of the ship, which represents how deep in the water it is. This is important as it is used in calculations for adjustments. The hopper sensors signal how much volume is in the hopper. These are significant because you need to know when the hopper can take its maximum capacity to reduce the total amount of trips you have to take. Finally, we also have the change in pressure draghead sensor, as illustrated in fig. 2. This sensor measures the pressure changes over the draghead.

Sensor failures in dredging can lead to serious consequences such as cavitation and sedimentation. Cavitation occurs when vacuum pressure drops too low, forming vapor bubbles that collapse and damage the pump. Sedimentation happens when the mixture velocity is too low, causing solid particles to settle in the pump, leading to blockages, abrasions, and reduced efficiency. Both issues can arise from inaccurate sensor readings like overreported velocity or density, which may lead to incorrect adjustments to pump operation, ultimately decreasing performance and increasing emissions.

A sensor can fail for different reasons, every sensor has a lifespan. Some may be in working condition for six months, whilst others can last for up to two years (O. van de Ven, personal communication, December 2024). This also depends on the conditions that the sensors are in. A sensors lifespan can be cut short due to harsh environmental conditions, such as storms. A pressure sensor could fail due to high pressures and get permanently malformed. Corrosion and

contamination can also cause damage on sensors. Over time, sensors also need to be recalibrated to make sure they work properly. In some, there could also be particles stuck on top or inside, for example sand grains. This can affect the sensor readings. After every trip, the dredgepump is flushed out to clear out debris from the pump which should reset the sensors. This is, however, not always the case, with some debris still influencing sensor readings after the cleansing.

2.2 Vacuum process

The system that we focus on monitors the vacuum process, and aims to increase productivity. This system uses the sensor signals of the velocity, vacuum pressure, density, and draught sensors. There is a governing equation that gives a prediction of the vacuum based on these inputs.

From a dredging handbook provided by Boskalis [57] and internal discussions, the vacuum formula whilst dredging is presented as:

$$V = \left(1 + \alpha + \xi + \lambda \frac{L}{D}\right) \frac{v^2}{2} \rho_m + (d_1 - d_2)\rho_m g - d_1 \rho_w g \tag{1}$$

where we have the variables defined as:

ρ_m: Mixture density (kg/m^3)
ρ_w: Water density (kg/m^3)
L: Length of pipe (m)
D: Diameter of pipe (m)
v: Velocity in pipe (m/s)
d_1: Water depth (m)
d_2: Pump depth (m)
g: Gravitational acceleration (m/s^2)
α: Entrance loss factor (dimensionless)
ξ: Sum of losses due to curves, hoses, etc. (dimensionless)
λ: Resistance factor in pipeline (dimensionless)

The formula shows that the pressure in the system must be in equilibrium. The pressure from the vacuum (left hand side) must be equal to the pressure from the right hand side. This formula can be split up in three different sections. The first part of the right hand side: $\left(1+\alpha+\xi+\lambda\frac{L}{D}\right)$ describes the pressure losses in the suction pipeline.

The entrance loss factor α is a parameter that measures the losses when the mixture enters through the draghead, at the suction mouth. When the draghead is buried deep in the soil, more pressure is needed to dredge the material. In contrast, when the draghead is loosely hovering above the soil, there is a weaker pull on the soil. The losses that incur depending on how the draghead is positioned is represented by this α. Whilst dredging, this factor fluctuates due to the repositioning of the draghead within the trip. It can also be that debris gets stuck within the draghead, affecting the parameter value.

The variable ξ represents the losses arising from the design of the pump. If the pump has more bends and curves, then there would be a larger loss in the pressure. This parameter can be estimated once and is then assumed to be constant the entire time, since it is solely based on the design of the vessel.

The resistance factor λ is also called the friction factor, this measures the resistance from the pipeline. The friction factor is based on the Darcy-Weissbach equation [57]:

$$\Delta h = \lambda\frac{Lv^2}{2Dg}$$

Where Δh is the friction loss in the head.

This friction factor also remains constant after it has been estimated once. The variable could gradually change over time, since the more the pump is used, the more damage there may be to the inside walls. This could cause a different resistance.

The next part of the equation: $(d_1-d_2)\rho_m g$, calculates the pressure required to lift the mixture from the draghead at depth d_2, to the pump level at depth d_1. The final part of the equation, $d_1\rho_w g$, represents the hydrostatic pressure

of the surrounding seawater at the draghead. The surrounding water pressure assists in driving the mixture into the suction pipe. As a result, the pump only needs to overcome the remaining pressure difference required to lift the mixture, after taking the pressure losses associated with α, ξ, and λ in account.

The majority of these variables such as the length and diameter of the pipes will always remain constant on a ship. Variables such as g remain the same no matter which circumstances, whilst the fluctuating factors such as the mixture density (ρ_m) are based on the sensor reading of that value. The variables we focus on are $\alpha, \xi,$ and λ.

If we solely focus on what happens at the draghead, there is another equation that describes the pressure equilibrium at this area (C. van den Berg, personal communication, February 2024):

$$\Delta \mathrm{P}_{\mathrm{Head}} = \rho_w g(d_1 - d_2) - \frac{1}{2}\rho_m v^2(1+\alpha) - h_{head}\rho_m g \tag{2}$$

Where $\Delta \mathrm{P}_{\mathrm{Head}}$ signifies the pressure changes at the draghead, and h_{head} is the height of the draghead with respect to the pipe. The reason why this variable is so important is because it has a direct relation with α, which is the parameter that fluctuates the most out of $\alpha, \xi,$ and λ.

To prevent future issues with ill-posed problems (non-unique solutions), we assign a new variable. We define: $\theta = \left(1 + \alpha + \xi + \lambda\frac{L}{D}\right)$, where θ is dependent on the depths and lengths of the pipes in different vessels. In practice, we assume that only α changes over time, while the other terms remain constant per vessel. The potential ill-posedness arises because increases in α, ξ, or $\lambda\frac{L}{D}$ yield mathematical indistinguishable effects; all enter the equation in the same way. Consequently, if they were to be estimated separately, multiple parameter combinations could reproduce the same model response, resulting in non-unique solutions. By combining them into a single parameter θ, this reparameterization removes ambiguity and ensures that the model is well-posed. Then, eq. (1) simplifies to:

$$V = \theta \frac{v^2}{2} \rho_m + (d_1 - d_2)\rho_m g - d_1 \rho_w g \tag{3}$$

Throughout this research, we aim to find a good estimate for θ. This way, we can make predictions and give insight to the accuracy of the sensors. Having accurate predictions for θ allows for the calculation of the optimal suction production point in real-time.

2.3 Suction production optimization

To diminish emissions, as well as increase productivity, we optimize the suction production process. We are given an equation to estimate the production on a TSHD vessel, which is given by [59]:

$$Prod = \frac{\rho_m - \rho_w}{\rho_s - \rho_w} \frac{\pi}{4} D^2 v \simeq K \left(\rho_m - \rho_w\right) v \tag{4}$$

For ρ_m, ρ_w, ρ_s being the mixture, water and situ densities, respectively. The variable D is the diameter of the pipe, whilst v is the mixture velocity and $K \in \mathbb{R}$ is a constant. Whilst the water density and the diameter of the pipe always remain the same, the in-situ density does not fluctuate a lot. The in-situ density refers to the density of the mixture that is being dredged before it enters the draghead, this is commonly taken to be $2000 kg/m^3$. We introduced $K = \frac{1}{\rho_s - \rho_w} \frac{\pi}{4} D^2$ to capture all the constant terms.

The typical production curve is shown in fig. 3. This shows a parabolic type relation between the mixture velocity and the production for two different vacuum pressures. The density lines in the figure indicate at which part of the curve the dredging is being done. Where, if the mixture density is lower than the optimum, it can be increased by adjusting the draghead, for example. Using the vacuum equation together with a prediction for θ, we can also determine where on the curve the dredging process is at that moment, and what has to be done to dredge closer to optimal production. It is important to note that a higher vacuum represents a higher optimized production, but also runs a higher

risk of cavitation occurring at high pressures.

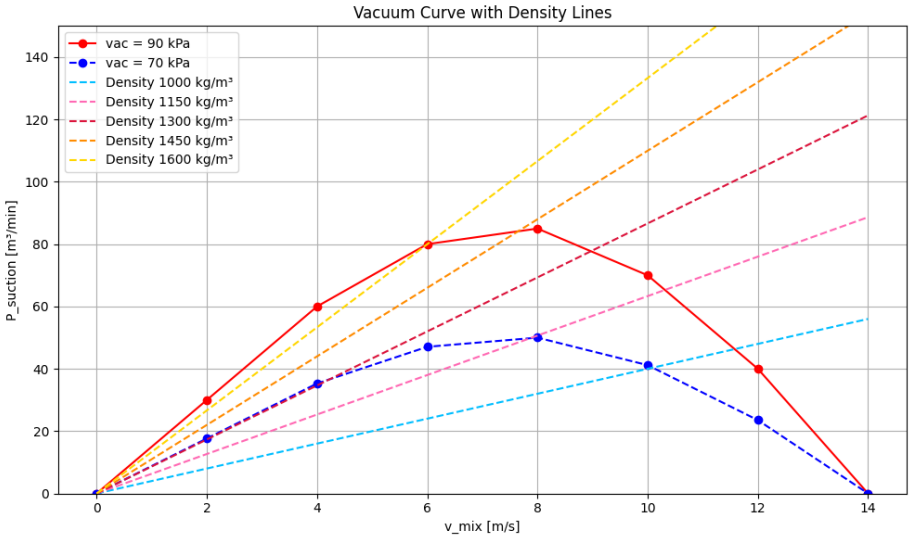


Figure 3: Production curve for varying flows, mixture densities and two different vacuums. Note that these curves are for illustrative purposes. In reality, the measured curves exhibit more variability and irregularities than the smooth representation shown here.

As can be seen in eq. (4) for production, there does not seem to be any quadratic behavior that causes this curve; the relation looks linear. The curve resembles a quadratic relation because we combine eq. (4) together with the vacuum formula (eq. (3)).

Rearranging the vacuum formula (eq. (3)) for the mixture density:

$$\rho_m(V, d_1, v, \theta, d_2) = \frac{V + d_1 \rho_w g}{\frac{\theta v^2}{2} + (d_1 - d_2) g}$$

Plugging this into our production equation (eq. (4)) we end up with:

$$Prod\left(m^3/s\right) = K\left(\rho_m(V, d_1, v, \theta, d_2) - \rho_w\right) v \tag{5}$$

By substituting the vacuum formula into the production equation, we obtain a formulation in which the mixture density is expressed as a function of the

vacuum, pipe depths, velocity, and θ. This allows the production to be written in terms of measurable and predictable quantities, rather than relying on delayed density measurements (which is explained in the next section).

In practice, this provides a two-step procedure. First, for a given predicted value of θ, a set of production curves can be constructed for different velocities. These curves represent the expected production levels, and identify the optimal production point. Second, by applying the same predicted θ to real-time measurements of the velocity, the current mixture density at the draghead can be inferred and the corresponding point on the production curve can be identified.

Locating the current operating point on this curve is valuable, as it enables an assessment of how far the dredging process is from the optimal production region. If the mixture density is too low, measures such as adjusting the draghead depth or angle can be advised to increase the density and move closer to the optimum. Conversely, if the density is too high, the operator can open the vacuum relief valve. This allows water to flow into the suction pipe, decreasing the mixture density. This way, the combined model does not only describe the relation between production, vacuum and mixture density, but also provides a practical tool for operators aiming for optimized production.

2.4 Delayed density sensor

Taking another look at the location of the different sensors in fig. 2, we see that both the mixture density and velocity are measured after the pump. The reasoning behind this is that these sensors are radioactive, so putting them closer to the water will result in a risk of leaking radioactive material into the sea. This does pose a challenge in the terms of the vacuum formula, however. As discussed in section 2.2, the vacuum formula expresses the balance between the pressure exerted by the pump and the pressure differences within the suction process. This balance is time-dependent: in order for the relation to hold, the sensor data must be considered at the same time instant. Since the fluid can be treated as incompressible [43], both the velocity and the vacuum pressure are

assumed to remain constant along the suction pipe. Consequently, placing the sensors at the end of the suction pipe does not introduce any inconsistencies for the application of the vacuum formula. Physically, this means that the pressure and velocity conditions measured after the pump are representative of the entire suction line, as the mixture cannot compress or expand along its path.

This assumption does not hold for the density measurement. The density of the mixture entering through the draghead does not necessarily match the density of the mixture after the pump. When applying the vacuum formula at locations along the suction pipe (i.e., upstream of the pump), the density measurement obtained downstream will exhibit a time delay of approximately ten seconds, depending on the mixture velocity. This highlights the importance of estimating the loss parameter θ. With knowledge of θ at any given time, the mixture density can be inferred at any desired location along the pipe. This capability, in turn, enables the determination of optimal suction production configurations in real-time. We illustrate this issue in figs. 4 and 5 below.

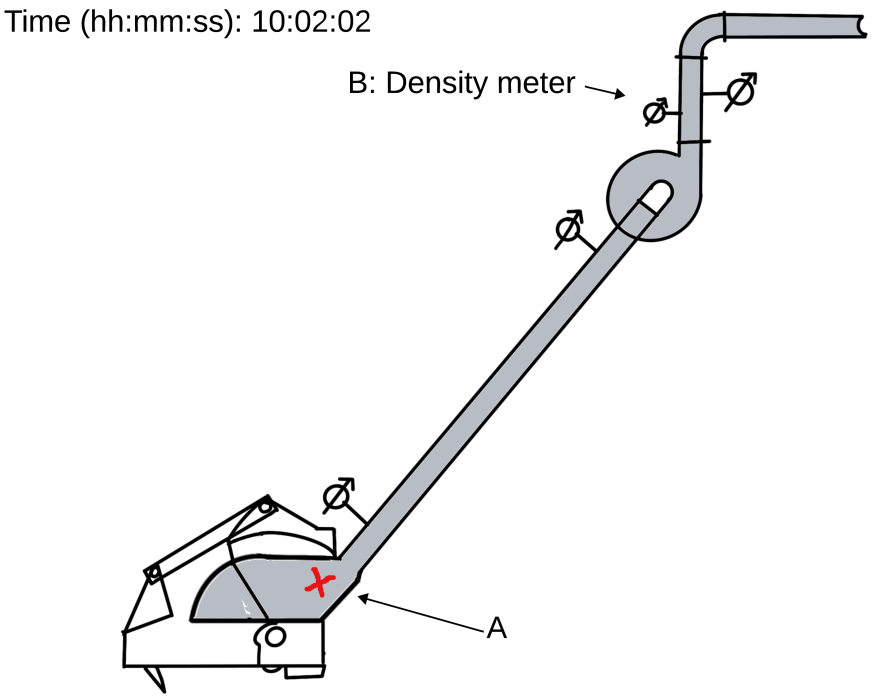


Figure 4: Dredgepump diagram with slurry going through it. We would like to know the mixture density at the draghead, at point A. The density sensor is located at point B, after the pump.

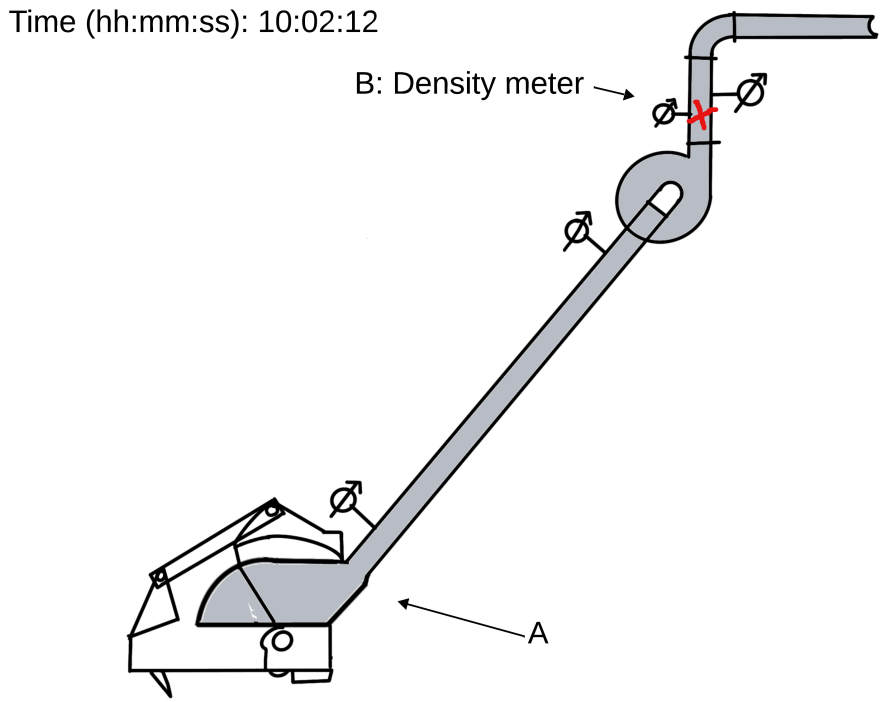


Figure 5: After approximately ten seconds, the density of the mixture that was at point A, is now measured at point B, where the sensor is located. Thus, we see that there is a delay in the density data, and currently no way to know the density in real-time.

3 Related Work

Dredging operations are complex systems where production efficiency rely heavily on accurate modeling [3]. Previous works have addressed production modeling in dredging using empirical and simulation-based methods. For example, approaches such as particle filtering to improve production efficiency on Trailing Suction Hopper Dredgers (TSHDs) have shown promise in real-time estimation contexts [65], while numerical studies have explored predicting the motion of the draghead to strengthen operating efficiency [9]. Deep learning approaches are also explored in combination with various classical machine learning through a stacking approach [3]. Probabilistic and statistical models were implemented to estimate productions of cutter suction dredgers [2] and bulldozers [46].

The issue of sensor instability and reliability is crucial in these systems. Faulty sensor data can undermine productivity and create conditions for mechanical collapse [29]. Li et al. applied several machine learning methods on sensor data of a TSHD to predict mechanical failure [29]. The approach provided a more stable environment for TSHD construction in the form of a digital twin-driven virtual sensor. Deep learning techniques are again utilized to design a prevention expert system from sensor data for online faults in cutter suction dredgers [56].

Physics-guided machine learning is an approach that incorporates physical knowledge into data-driven models in order to improve accuracy and interpretability. The core idea is to combine the flexibility of machine learning with the domain insights provided by physical laws [44]. Such methods have proven particularly valuable in engineering and environmental systems, where data is often limited and physical consistency is essential [64]. Some works embed physics into the network architecture itself, such as Daw et al. in lake temperature modeling [11] and Mohajer et al. for multistep predictions of aerial vehicles [38]. Other articles rely on physics-informed preprocessing to prepare more structured input data, as in [55, 48, 47]. Another method involves physics-

based regularization terms, prominently explored in Physics-Informed Neural Networks (PINNs) [45] for solving forwards and inverse problems. In the context of an inverse problem, PINNs have been extended to approximate time varying parameters for a range of different applications [36, 71, 18]. Together, these approaches demonstrate the diverse ways in which physical knowledge can be embedded into machine learning workflows.

Operator learning has emerged as a promising direction for learning mappings between infinite-dimensional function spaces [35]. Operator learning is concerned with the approximation of operators, which are mappings from one function space to another, where the inputs and outputs are themselves functions [16]. The DeepONet framework [35] was among the first to demonstrate this research, showing strong performance in modeling complex operators, particularly for partial differential equations (PDEs). The initial architecture has since been extended to account for multiple input functions [21], as well as for information fusion of different layers [61]. Hybrid models that fuse PINNs and DeepONets, like those in [33, 60, 32], aim to combine operator learning with physical constraints. Other approaches embed physics without relying on PDE loss functions, as in [67] through non-linear interactions of branch and trunk outputs, and in [39] with Fourier-enhanced architectures.

Given the dependence on sensors for operational control, anomaly detection and correction have become increasingly relevant [56]. Techniques like wavelet-based fault detection [15], sensor drift identification [20], and abrupt-change monitoring [72] have been used to detect sensor outliers. The combination of sensor data and production optimization in real-time has been applied in numerous industries such as reservoir management and oil production [42, 6, 19].

To address time-varying behavior and sensor error propagation, correction models using online learning and transfer learning have gained traction [23, 8]. Transfer learning has been implemented together with DeepONets [66] and PINNs [12, 8], showcasing adaptability across domains. Online correction strategies, ranging from air quality forecasting [1] to hydrological predictions [51, 73] and traffic flow [23] highlight the benefits of residual modeling to calibrate pre-

existing models. These efforts suggest potential for lagged value correction models in the dredging domain, especially when paired with operator learning or PINN frameworks.

While significant progress has been made in production modeling [65], physics-guided machine learning [11, 38], and sensor fault detection [42, 6, 19] across related engineering domains, research gaps remain in the context of dredging. Existing production models for TSHDs largely rely on empirical relations or simulation studies [65, 9], which limits their generalizability to real operations. Machine learning methods have been tentatively explored in the dredging domain [3], physics-guided approaches, however, remain largely absent. Yet, physics-guided machine learning offers a promising route toward improved interpretability and robustness [44]. At the same time, operator learning has shown strong potential for approximating complex mappings [35], but its integration with physics-based constraints in real-time dredging settings remains underdeveloped. Furthermore, sensor reliability has been studied through digital twins [29], wavelet-based detection [15], and transfer learning [8]. The challenge of handling noisy measurements and exploiting simulation-trained models, all whilst providing reliable production estimates has not yet been addressed, particularly in dredging domains. These gaps motivate the research questions below. These focus on predicting the loss parameter θ, incorporating physics into operator learning, designing correction mechanisms for sensor noise, and enabling anomaly detection throughout the vacuum dredging process.

We formulate the main research question as:

To what extent can the loss parameter θ be accurately predicted using simulated datasets, and how well does this generalize to real-world sensor data?

This question is explored through several supporting sub-questions:

1. **How can operator learning architectures be designed to incorpo-**

rate physics-based domain knowledge in order to improve prediction accuracy and provide model interpretability?

2. **In what ways can lagged sensor values be incorporated as correction mechanisms to progressively improve model outputs?**

3. **At which stages of the vacuum dredging process can sensor health be monitored, and how can the model support reliable anomaly detection in real-time?**

4. **To what extent can a hybrid architecture, utilizing simulated data for training in combination with correction modules, improve the robustness of suction production estimates in the presence of sensor noise?**

4 Methodology

4.1 Technical framework

In this section, we discuss the technical details attached to the methods we use. We start by giving a brief introduction to neural networks and transformers in sections 4.1.1 and 4.1.2, then we go deeper into specific operator learning architectures in section 4.1.3. This gives the basis for the designed architecture we propose in section 4.2.

4.1.1 Neural networks

Neural networks [37] are computational models inspired by the structure of the brain, consisting of layers of interconnected nodes (neurons). We consider a dataset of n input–output pairs $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$ with inputs $x_i \in \mathbb{R}^d, \quad x_i = (x_{i1}, x_{i2}, \ldots, x_{id})$ and corresponding outputs $y_i \in \mathbb{R}^m$. The goal of a neural network is to approximate a mapping $f : \mathbb{R}^d \to \mathbb{R}^m$ such that $\hat{y}_i = f(x_i) \approx y_i$.

The simplest case is the perceptron, which computes its prediction through a linear transformation followed by a (possibly nonlinear) activation function α:

$$\hat{y}_i = \alpha\left(\sum_{j=1}^{d} w_j x_{ij} + b\right),$$

where $w_j \in \mathbb{R}, \quad j = 1, \ldots, d$ are the weights associated with each input dimension, $b \in \mathbb{R}$ is the bias, and $\hat{y}_i \in \mathbb{R}$ is the output for input x_i.

Weights w_i determine input importance, and the bias b shifts the output. The variable $\hat{y}_i$ represents the networks prediction. The parameter α is an activation function such as ReLU or sigmoid [13]. These are essential for the network to learn something other than linear relationships. Throughout the project we make use of the hyperbolic tangent activation function. The plot of which is shown in fig. 6.

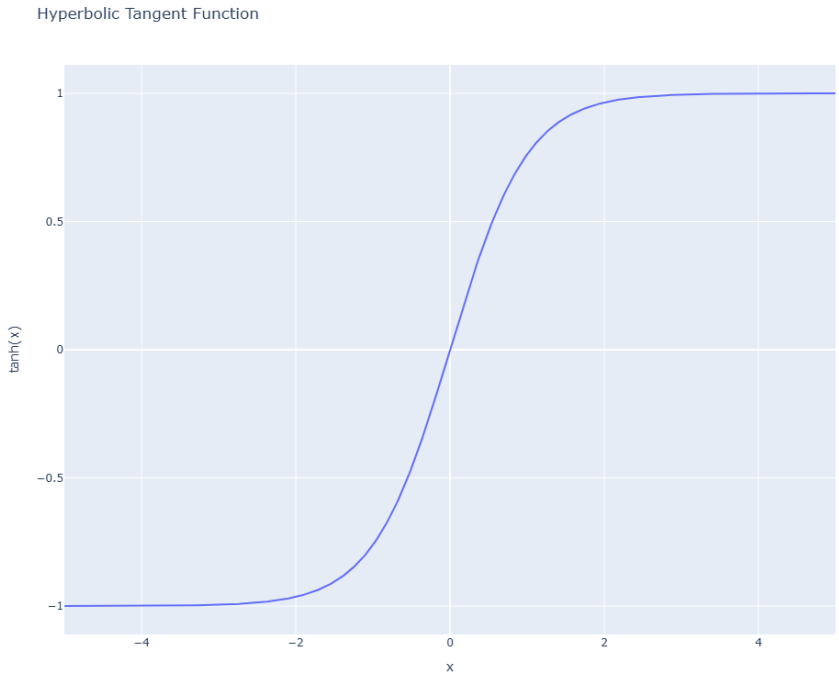


Figure 6: Hyperbolic tangent function: $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ with an output between $[-1, 1]$. It helps neural networks capture both positive and negative activations because it is centered around zero and smoothly differentiable [69].

The network's parameters: its weights and biases, are adjusted during training to minimize a loss function such as the mean squared error (which is the loss function that we will use during training). For a function f, which could learned be a neural network, and dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, we define the mean squared error as:

$$\mathcal{L}(f, \mathcal{D}) = \frac{1}{n} \sum_{i=1}^{n} \|f(x_i) - y_i\|^2 ,$$

where $\|\cdot\|$ denotes the Euclidean norm in $\mathbb{R}^m$.

Extending this idea, a multilayer perceptron (MLP) introduces one or more hidden layers between the input and the output layer. Each hidden layer consists of several neurons, where each neuron applies a weighted sum of its inputs followed by a nonlinear activation function.

Let the first hidden layer contain H_1 neurons. For an input $x_i \in \mathbb{R}^d$, the pre-activation of neuron k in this hidden layer is

$$Z_{ik}^{(1)} = \sum_{j=1}^{d} w_{kj}^{(1)} x_{ij} + b_k^{(1)}, \quad k = 1, \ldots, H_1,$$

with weights $w_{kj}^{(1)} \in \mathbb{R}$ and biases $b_k^{(1)} \in \mathbb{R}$. Applying the activation function α gives the hidden representation

$$H_{ik}^{(1)} = \alpha\Big(z_{ik}^{(1)}\Big), \quad H_i^{(1)} \in \mathbb{R}^{H_1}$$

This process repeats for subsequent hidden layers. For a network with L layers (where layers $1, \ldots, L-1$ are hidden and layer L is the output layer), the forward pass can be written recursively:

$$Z_i^{(\ell)} = W^{(\ell)} H_i^{(\ell-1)} + b^{(\ell)}, \quad H_i^{(\ell)} = \alpha\big(Z_i^{(\ell)}\big), \quad \ell = 1, \ldots, L-1,$$

where

$$W^{(\ell)} \in \mathbb{R}^{H_\ell \times H_{\ell-1}}, \quad b^{(\ell)} \in \mathbb{R}^{H_\ell},$$

are the weight matrices and bias vectors, respectively. Here, $H_i^{(0)} = x_i \in \mathbb{R}^d$. Finally, the output layer computes:

$$Z_i^{(L)} = W^{(L)} H_i^{(L-1)} + b^{(L)}, \quad \hat{y}_i = \alpha\big(Z_i^{(L)}\big),$$

where

$$W^{(L)} \in \mathbb{R}^{m \times H_{L-1}}, \quad b^{(L)} \in \mathbb{R}^m, \quad \hat{y}_i \in \mathbb{R}^m$$

Thus, the complete feedforward neural network is a composition of transformations:

$$\mathcal{NN}^{w,b} = \hat{y}_i = f(x_i) =$$
$$\alpha\Big(W^{(L)}\, \alpha\big(W^{(L-1)}\, \alpha(\ldots \alpha(W^{(1)} x_i + b^{(1)}) \ldots) + b^{(L-1)}\big) + b^{(L)}\Big)$$

Training a neural network involves minimizing the loss function

$$\mathcal{L}(\mathcal{NN}^{w,b}, \mathcal{D})$$

This is minimized by optimizing the weights w and biases b within all the layers.

The optimization process consists of several keys stages. The first is weight initialization, where the weights are set using random values from predefined distributions or other methods [17]. Proper initialization helps ensure effective gradient flow and avoids premature convergence [40]. After this, input data is passed through each layer of the network where successive linear transformations and activation functions are applied to produce the model's prediction. Then, the network computes gradients of the loss function with respect to each parameter by applying the chain rule in a recursive manner. This is called back-propagation [63]. The computed gradients are used to update the network's parameters via an optimization algorithm, such as gradient descent [53].

For a given weight $w_{kj}^{(l)}$, the gradient descent update rule takes the form:

$$w_{kj}^{(l)} \leftarrow w_{kj}^{(l)} - \mu \nabla w_{kj}^{(l)},$$

where μ is the learning rate that controls the step size of each update.

In our case, we employ a variation of gradient descent: Adaptive Moment Estimation (adam) [24]. Adam improves the basic update rule by incorporating an exponentially decaying average of past gradients (first moment) and an exponentially decaying average of past squared gradients (second moment). Denoting the gradient at time step t as g_t, these estimates are:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

with bias-corrected forms

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

The parameter update then becomes

$$w_{kj}^{(l)} \leftarrow w_{kj}^{(l)} - \mu \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

where $\beta_1, \beta_2 \in [0, 1)$ control the decay rates and ϵ is a small constant for numerical stability.

This cycle repeats iteratively until a convergence criterion is met or a predefined stopping condition is satisfied. This is when, for example, the training loss has reached a certain threshold.

One of the main reasons why neural networks are so widely used in the literature over the past years is due to the fact that neural networks are universal function approximators [53]:

Theorem 1 (Universal approximation theorem for neural networks)**.** *Let $K \subset \mathbb{R}^d$ be a compact set and let $g : K \to \mathbb{R}^m$ be a continuous function in K. Then, for every $\varepsilon > 0$, there exists a feedforward neural network*

$$f : \mathbb{R}^d \to \mathbb{R}^m,$$

with a single hidden layer, weight matrices $W^{(\ell)}$ and bias vectors $b^{(\ell)}$ such that

$$\sup_{x \in K} \left\| f(x) - g(x) \right\| < \varepsilon \text{ for all } x \in K$$

In other words, feedforward neural network f with one hidden layer and a nonlinear activation function can approximate function g with a degree of accuracy ε, given sufficient number of neurons in the hidden layer.

We can see that from theorem 1, given sufficient neurons and suitable activation functions, a feedforward neural network can approximate any continuous function arbitrarily well.

4.1.2 Transformers

A type of architecture stemming from neural networks is the transformer, which was originally introduced for sequence modeling and has since become one of the most influential deep learning architectures [58]. While neural networks learn through layer-wise transformations and gradient-based optimization, transformers build upon the same principles but introduce a more flexible way to model relationships between inputs. Instead of relying solely on fixed connections between layers, they use an attention mechanism that dynamically determines which inputs should influence each other. This allows the model to weigh the importance of different input elements relative to each other.

Given an input sequence $X = [x_1, x_2, \ldots, x_T]$, each element x_i is first projected into three vectors: a *query* $q_i = W_Q x_i$, a *key* $k_i = W_K x_i$, and a *value* $v_i = W_V x_i$, where W_Q, W_K, W_V are learnable weight matrices. The attention output for each element is then computed as a weighted combination of all value vectors:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V,$$

where Q, K, V are matrices collecting all queries, keys, and values, and d_k is the dimensionality of the key vectors used for normalization.

The softmax function converts the raw similarity scores into normalized attention weights. For a vector $z = [z_1, z_2, \ldots, z_n] \in \mathbb{R}^n$, it is defined as

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}, \quad i = 1, \ldots, n$$

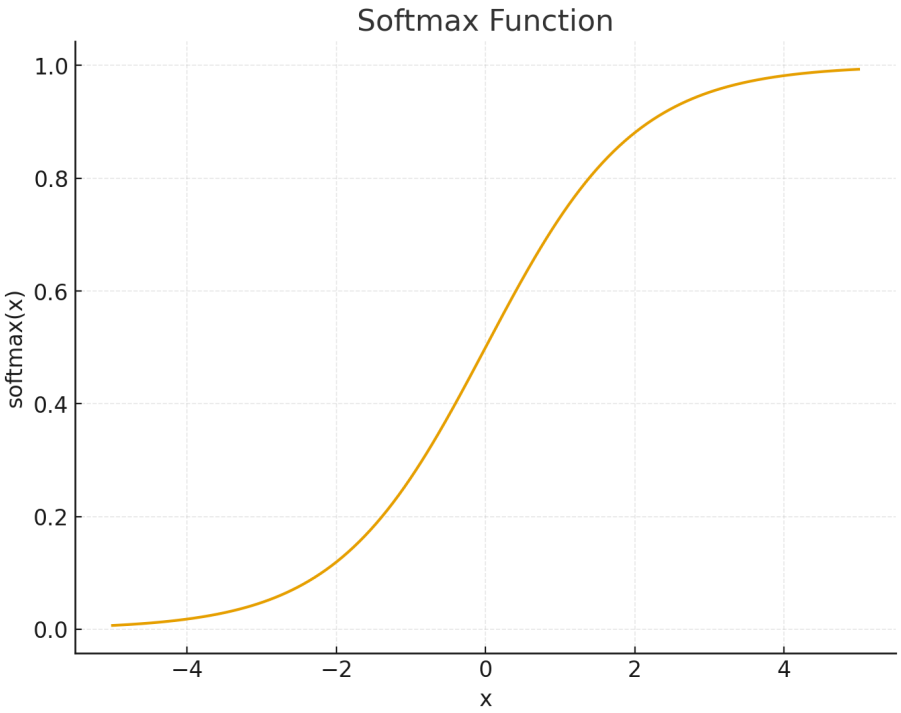


Figure 7: Softmax function, shown here for a two-class case where the second logit is set to zero. The output lies in the range $[0, 1]$ and represents normalized class probabilities that sum to one [22].

The softmax formula ensures that all weights are positive and sum to one, effectively forming a probability distribution over the input elements. Overall, transformers preserve the same optimization dynamics as traditional neural networks but enhance them through attention, enabling more expressive modeling of dependencies within the data. This concept motivates the attention-based interactions introduced later in our proposed architecture.

4.1.3 Operator learning

In contrast to neural networks being universal approximators of functions as shown in theorem 1, it has been shown that a neural network with a single hidden layer can also approximate any nonlinear operator with arbitrary accuracy [7]. Such an operator represents a mapping between infinite-dimensional function

spaces, extending the universal approximation property to functional inputs and outputs (see theorem 2).

To implement this theorem into practice, DeepONets (among other architectures such as Fourier Neural Operators [30] and Graph Neural Operators [31]) have been introduced in [35].

The idea is that the DeepONet consists of two subnetworks: a branch and a trunk network. The branch network processes an input function $f : \mathbb{R}^{d_x} \to \mathbb{R}^{d_f}$, sampled at n discrete sensor locations $\{x_1, x_2, \ldots, x_n\}$, where each $x_i \in \mathbb{R}^{d_x}$. These samples form the input vector $\{f(x_1), f(x_2), \ldots, f(x_n)\} \in \mathbb{R}^{nd_f}$.

The trunk network processes evaluation points $z \in \mathbb{R}^{d_z}$, where the operator's output is evaluated. Each subnetwork encodes its respective input into a latent feature representation, $(b_1, \ldots, b_p) \in \mathbb{R}^p$ for the branch and $(t_1, \ldots, t_p) \in \mathbb{R}^p$ for the trunk. These representations are then combined to produce the operator output

$$\mathcal{G}(f)(z) \in \mathbb{R}^{d_y}$$

The operator $\mathcal{G}$ thus defines a mapping that takes an input function f and produces a corresponding output function evaluated at z.

Here, d_x and d_z denote the dimensionality of the input and output domains, respectively; d_f and d_y represent the dimensions of the input and output function values; n is the number of sensor points at which f is sampled; and p is the size of the feature space shared between the branch and trunk networks. In most applications, $d_f = d_y = 1$, corresponding to scalar-valued functions.

Lu et al. [35] discuss both a stacked (fig. 8a) and unstacked (fig. 8b) DeepONet architecture. In the stacked version, multiple branch networks are used to encode the input function, while in the unstacked version a single branch network encodes the input function f. We focus on the unstacked case.

For this case, the branch network takes discrete samples of the input function,

$$\{f(x_1), f(x_2), \ldots, f(x_n)\}$$

and encodes them into a feature vector $(b_1, \ldots, b_p) \in \mathbb{R}^p$:

$$\{f(x_1), f(x_2), \ldots, f(x_n)\} \underset{\text{branch network}}{\longrightarrow} (b_1(f), \ldots, b_p(f))$$

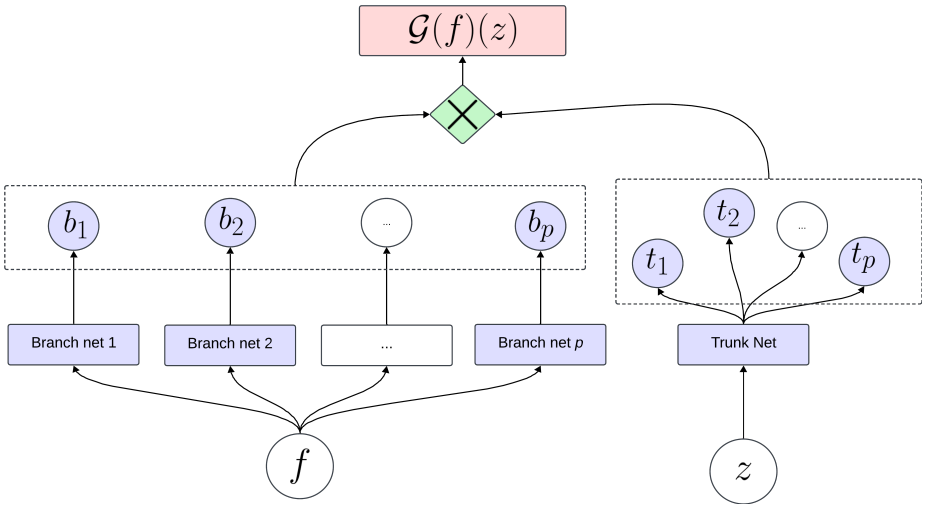
The trunk network, on the other hand, takes evaluation points z as input and outputs a set of basis functions $(t_1, \ldots, t_p) \in \mathbb{R}^p$:

$$z \underset{\text{trunk network}}{\longrightarrow} (t_1, \ldots, t_p)$$

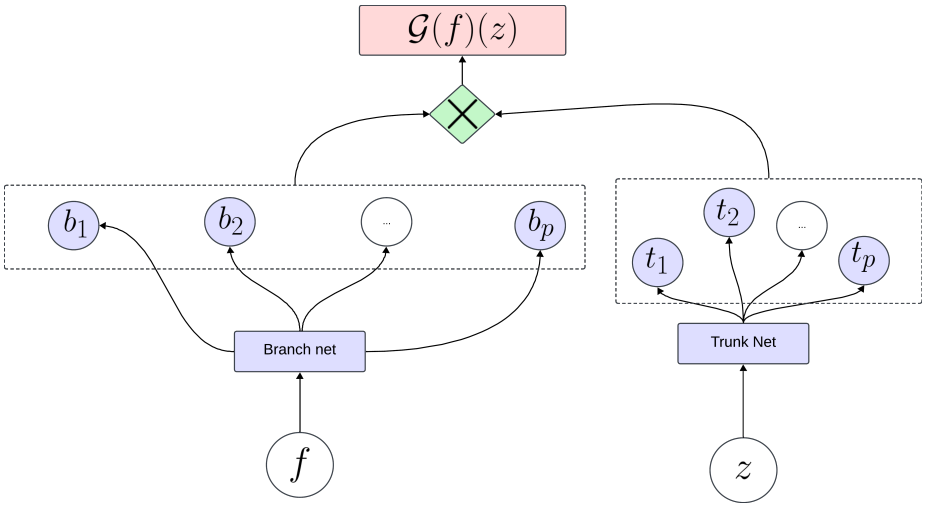
The branch features (b_k) depend on the input function f, while the trunk features (t_k) depend only on the evaluation points z. They are therefore not fixed coefficients but vary with their respective inputs.

Finally, the operator output is obtained by combining the two representations through an inner product (represented by the green box in fig. 8):

$$\mathcal{G}(f)(z) \simeq \sum_{k=1}^{p} b_k \, t_k$$



(a) Stacked DeepONet, taken from [35].



(b) Unstacked DeepONet, taken from [35].

Figure 8: Schematic of the the stacked and unstacked DeepONet.

We present the theorem in [35] as:

Theorem 2 (Universal approximation theorem for operators)**.** *Suppose that α is a continuous nonpolynomial activation function, X is a Banach space, $K_1 \subset X$ and $K_2 \subset \mathbb{R}^{d_z}$ are compact sets in X and $\mathbb{R}^{d_z}$, respectively, V is a compact subset of $C(K_1)$, and $\mathcal{G}$ is a nonlinear continuous operator that maps*

V into $C(K_2)$. Then, for any $\varepsilon > 0$, there exist positive integers L_b, p, and n, constants $w_{ijk}^{(b)}$, $b_{ik}^{(b)}$, $w_k^{(t)}$, $b_k^{(t)}$, $C_{ik} \in \mathbb{R}$, and sensor locations $x_j \in K_1$, for $i = 1, \ldots, L_b$, $k = 1, \ldots, p$, and $j = 1, \ldots, n$, such that

$$\left| \mathcal{G}(f)(z) - \sum_{k=1}^{p} \underbrace{\sum_{i=1}^{L_b} C_{ik}\, \alpha\left(\sum_{j=1}^{n} w_{ijk}^{(b)}\, f(x_j) + b_{ik}^{(b)} \right)}_{branch} \underbrace{\alpha\left(w_k^{(t)} \cdot z + b_k^{(t)} \right)}_{trunk} \right| < \varepsilon$$

holds for all $f \in V$ and $z \in K_2$.

This formalizes the idea that neural networks can approximate nonlinear operators to arbitrary accuracy. It demonstrates that the operator $\mathcal{G}$; mapping between function spaces, can be represented using two subnetworks: a branch network and a trunk network. The inner summation corresponds to the branch network, which acts on the sampled input function values $\{f(x_j)\}_{j=1}^{n}$ using trainable weights $w_{ijk}^{(b)}$ and biases $b_{ik}^{(b)}$. The coefficients C_{ik} serve as output-layer weights of the branch network, linearly combining the activations indexed by i to form the branch features b_k. Similarly, the trunk network acts on the evaluation points z through its own trainable weights $w_k^{(t)}$ and biases $b_k^{(t)}$. Both subnetworks apply a nonlinear activation function α, enabling them to capture complex, non-linear relationships in their respective inputs. The outputs of the branch and trunk networks form feature representations, which are combined through an inner product:

$$\sum_{k=1}^{p} b_k\, t_k,$$

producing an approximation of the operator output $\mathcal{G}(f)(z)$. The theorem guarantees that with sufficient neurons and suitable activation functions, such a composition of neural networks can approximate any continuous nonlinear operator within an arbitrary precision ε.

These DeepONets have been applied to solve problems related to partial differential equations (PDEs) [35]. A DeepONet can be used to learn the solution operator of a PDE system. Assuming we have a PDE with solution $u(x, t)$

that depends on some input function $f(x)$ (for instance, a boundary or initial condition), the relationship between them can be represented as an operator

$$\mathcal{G} : f \mapsto u = \mathcal{G}(f),$$

so that $u(x,t) = \mathcal{G}(f)(x,t)$. After the network learns this operator, it can then predict the solution $u(x,t)$ across spatial and temporal coordinates of interest, given the input function $f(x)$, without resolving the entire PDE.

There have been numerous modifications and extensions to the DeepONet, with one architecture accounting for multiple inputs: the multiple input operator network (MIONet) [21]. A naive approach to handle multiple input functions would be to concatenate the functions together to act as the input of a branch network in the DeepONet. The approach of the MIONet has shown to outperform the concatenation method. The main result in [21] was extending the universal approximation theorem for operators (theorem 2) from one Banach space X, to a product of multiple Banach spaces $X_1, \dots X_m$. This way, the input of the operator can be expanded to multiple functions. The MIONet still works with branch and trunk networks, with the trunk network remaining unchanged. For m input functions $f_1, \dots f_m$, at n sensor locations: $\{f_i(x_1), f_i(x_2) \dots f_i(x_n)\}\ \forall i$, we configure m independent branch networks. With this methodology, every input function has its own corresponding branch network. Combining the output of all the branch networks together with the trunk network with input z results in the operator $\mathcal{G}(f_1, \dots f_m)(z) \in \mathbb{R}$ (see fig. 9). Operator $\mathcal{G}$ now takes multiple functions as input, instead of just one as in the classical DeepONet methodology.

Before diving into the extended universal approximation theorem: theorem 3; we present a preliminary definition and property (taken from [21]):

Definition 1 (Schauder Basis)**.** *Let X be an infinite-dimensional normed linear space. A sequence $\{e_i\}_{i=1}^{\infty}$ in X is called a Schauder basis of X, if for every $x \in X$ there is a unique sequence of scalars $\{a_i\}_{i=1}^{\infty}$, called the coordinates of x,*

such that

$$x = \sum_{i=1}^{\infty} a_i e_i$$

We denote the coordinate functionals of the Schauder basis elements e_i, by e_i^*. These coordinate functionals extract the coordinate of x along the basis vector e_i. That is,

$$x = \sum_{i=1}^{\infty} e_i^*(x) e_i, \forall x \in X$$

Property 1 (Canonical Projection)**.** *Assume that K is a compact set in a Banach space X equipped with a Schauder basis $\{e_i\}_{i=1}^{\infty}$ and corresponding coordinate functionals $\{e_i^*(x)\}_{i=1}^{\infty}, x \in X$. For a canonical projection P_n, we have*

$$\lim_{n\to\infty} \sup_{x\in K} \|x - P_n(x)\| = 0$$

The P_n is decomposed as

$$P_n = \psi_n \circ \varphi_n,$$

where $\varphi_n : X \to \mathbb{R}^n$ and $\psi_n : \mathbb{R}^n \to X$ are defined as

$$\varphi_n(x) = (e_1^*(x), \cdots, e_n^*(x))^T, \quad \psi_n(\alpha_1, \cdots, \alpha_n) = \sum_{i=1}^{n} \alpha_i e_i$$

The $\varphi_n(x)$ represent the truncated coordinates for x.

We present the universal approximation theorem for multiple Banach spaces [21]:

Theorem 3 (Universal approximation theorem for multiple Banach spaces)**.** *Suppose that α is a continuous nonpolynomial activation function, $X_1, \ldots, X_m, Y$ are Banach spaces, $K_i \subset X_i$ are compact sets for each $i = 1, \ldots, m$, and each X_i has a Schauder basis with canonical projections $P_i^{(q_i)} = \psi_i^{(q_i)} \circ \varphi_i^{(q_i)}$. Assume that $\mathcal{G} : K_1 \times \cdots \times K_m \to Y$ is a continuous operator. Then, for any $\varepsilon > 0$, there exist positive integers L_b, p_i, q_i, constants $w_{r\ell k_i}^{(b_i)}$, $b_{rk_i}^{(b_i)}$, $w_{k_1,\ldots,k_m}^{(t)}$, $b_{k_1,\ldots,k_m}^{(t)}$, $C_{rk_1\ldots k_m} \in \mathbb{R}$, and sensor locations $x_\ell^{(i)} \in K_i$, for $r = 1, \ldots, L_b$, $\ell = 1, \ldots, q_i$, and $k_i = 1, \ldots, p_i$, such that:*

for each $i = 1, \ldots, m$, we denote the branch pre activation output as:

$$b_{r,k_i}^{(i)}(f_i) \coloneqq \sum_{\ell=1}^{q_i} w_{r\ell k_i}^{(b_i)} \, (\varphi_i^{(q_i)}(f_i))_\ell \; + \; b_{rk_i}^{(b_i)}$$

Such that we can define the branch features for the indices $(k_1, \ldots, k_m)$ as:

$$B_{k_1,\ldots,k_m}(f_1, \ldots, f_m) \coloneqq \sum_{r=1}^{L_b} C_{rk_1\ldots k_m} \, \alpha\left(\sum_{i=1}^{m} b_{r,k_i}^{(i)}(f_i) \right)$$

The features coming from the trunk network are given as:

$$T_{k_1,\ldots,k_m}(z) \coloneqq \alpha\Big(w_{k_1,\ldots,k_m}^{(t)} \cdot z + b_{k_1,\ldots,k_m}^{(t)} \Big)$$

Then we have that

$$\left\| \mathcal{G}(f_1, \ldots, f_m) - \sum_{k_1=1}^{p_1} \cdots \sum_{k_m=1}^{p_m} B_{k_1,\ldots,k_m}(f_1, \ldots, f_m) \, T_{k_1,\ldots,k_m}(z) \right\| < \varepsilon$$

for all $f_i \in K_i$ and $z \in Y$.

Then, theorem 3 provides the theoretical foundation for constructing the MIONet architecture. It formalizes that operator $\mathcal{G} : K_1 \times \cdots \times K_m \to Y$, acting on multiple input functions, can be approximated by a composition of neural subnetworks: m branch networks and one trunk network. Each branch network processes its respective input function $f_i \in K_i$, sampled through its basis projection $\varphi_i^{(q_i)}(f_i)$. The branch term:

$$\alpha\left(\sum_{i=1}^{m} \sum_{\ell=1}^{q_i} w_{r\ell k_i}^{(b_i)} \, (\varphi_i^{(q_i)}(f_i))_\ell + b_{rk_i}^{(b_i)} \right)$$

corresponds to the nonlinear transformation performed by the m branch networks, parameterized by weights $w_{r\ell k_i}^{(b_i)}$ and biases $b_{rk_i}^{(b_i)}$. The coefficients $C_{rk_1\ldots k_m}$ act as output-layer weights, linearly combining the activations of the branch subnetworks to form joint branch features indexed by $(k_1, \ldots, k_m)$.

This formulation of the branch features corresponds to the high-rank version

of the MIONet, where $B_{k_1,\ldots,k_m}$ is formed by a nonlinear combination of all branch pre-activations across the m input functions. In practice, a low-rank version of MIONet is typically adopted to reduce the number of parameters and computational cost [21]. In this setting, the combined branch feature is defined directly in terms of the branch activations at the final layer as the elementwise (Hadamard) product of the individual branch representations:

$$B_{k_1,\ldots,k_m}(f_1,\ldots,f_m) \;=\; \prod_{i=1}^{m} \alpha\left(\sum_{\ell=1}^{q_i} w_{\ell k_i}^{(b_i)}\,(\varphi_i^{(q_i)}(f_i))_\ell + b_{k_i}^{(b_i)}\right),$$

The resulting operator then also takes the form

$$\mathcal{G}(f_1,\ldots,f_m)(z) \;=\; \sum_{k_1=1}^{p_1}\cdots\sum_{k_m=1}^{p_m} B_{k_1,\ldots,k_m}(f_1,\ldots,f_m)\,T_{k_1,\ldots,k_m}(z) \;+\; b,$$

which corresponds to the default low-rank MIONet architecture proposed by Lu et al.[21]. Parameter $b \in \mathbb{R}$ is an additional bias at the final step to help with training [21]. The low-rank version is favored in most applications as it provides a more compact representation of the operator while maintaining sufficient expressive power.

The trunk term:

$$\alpha\Big(w_{k_1,\ldots,k_m}^{(t)} \cdot z + b_{k_1,\ldots,k_m}^{(t)}\Big)$$

represents the feature mapping of the evaluation points $z \in Y$ through the trunk network, which has its own learnable weights and biases. Both subnetworks use the nonlinear activation function α to capture complex dependencies within their respective inputs.

The outputs of the branch ($B_{k_1,\ldots,k_m}(f_1,\ldots,f_m)$) and trunk ($T_{k_1,\ldots,k_m}(z)$) subnetworks are then coupled through a multilinear combination across all indices $(k_1,\ldots,k_m)$ (represented by the cross illustration in fig. 9), producing an approximation of the operator output $\mathcal{G}(f_1,\ldots,f_m)(z)$. This theorem guarantees that, with sufficient capacity and an appropriate activation function, such a MIONet architecture can approximate any continuous nonlinear operator be-

tween multiple Banach spaces to arbitrary accuracy ε.

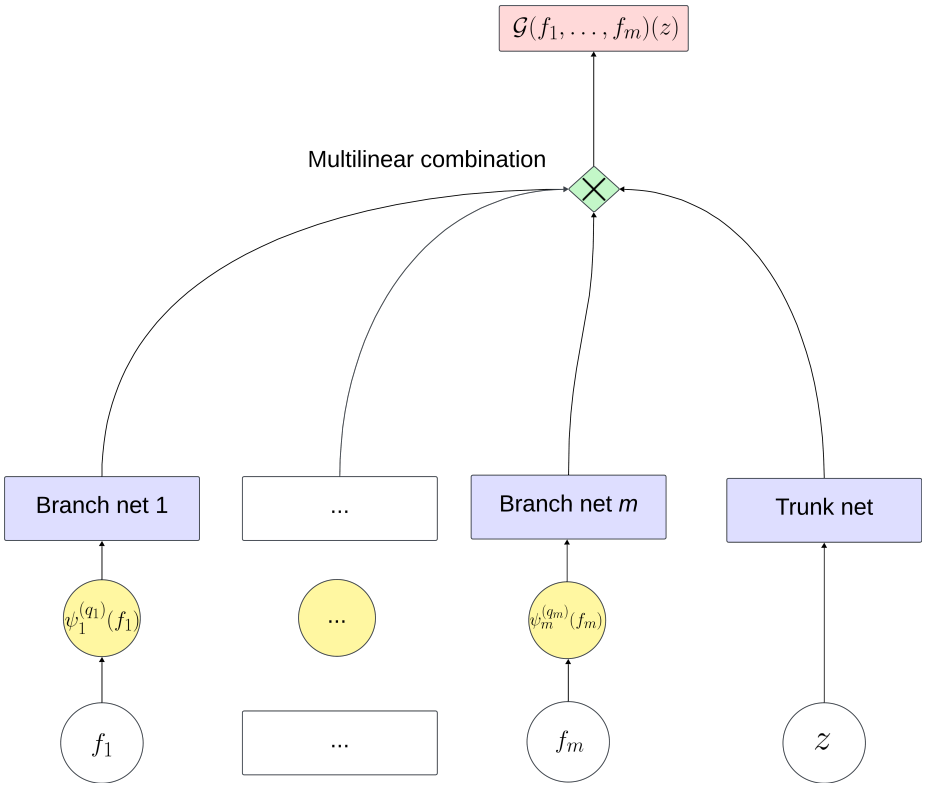


Figure 9: MIONet, independent branch network outputs are combined linearly with the trunk network output. The combination is illustrated by the green cross, corresponding to $\sum_{k_1=1}^{p_1} \cdots \sum_{k_m=1}^{p_m} B_{k_1,\ldots,k_m}(f_1, \ldots, f_m)\, T_{k_1,\ldots,k_m}(z) + b$ for the branch and trunk features, respectively. Adapted from [21].

An improved architecture to a regular DeepONet has been proposed in [61]. In the previous two architectures, it can be seen that the branch outputs are combined linearly, which is then also linearly combined with the trunk output. The improved architecture in [61] implements encoders for the branch and trunk networks such that non-linearity in input signals can be discovered. Both sub-networks interact with each other through multiple point-wise multiplications, which lead to more accurate results when compared to the original DeepONet. The author [61] argues that this is due to the higher resilience to vanishing signals.

We go back to the situation of the unstacked DeepONet (fig. 8b) where the input to the singular branch network is one function f, and the input to the trunk network is z. Simultaneously that the inputs go through a first hidden layer in the neural network, they also pass through an encoding layer (the yellow block in fig. 10).

We denote the weights and biases of the branch encoder as $w_{ij}^{(b,0)}$ and $b_i^{(b,0)}$, where $i = 1, \ldots, p$ indexes the neurons of the encoding layer and $j = 1, \ldots, n$ indexes the sensor locations of the input function f. Similarly, the weights and biases of the trunk encoder are denoted by $w_i^{(t,0)}$ and $b_i^{(t,0)}$.

The encoder block is then given by

$$U_i = \alpha\Big(\textstyle\sum_{j=1}^{n} w_{ij}^{(b,0)}\, f(x_j) + b_i^{(b,0)}\Big), \qquad V_i = \alpha\Big(w_i^{(t,0)} \cdot z + b_i^{(t,0)}\Big), \quad i = 1, \ldots, p$$

where $\alpha(\cdot)$ denotes the activation function.

These encoded vectors $U = (U_1, \ldots, U_p)$ and $V = (V_1, \ldots, V_p)$ constitute the encoding block (yellow in fig. 10), which transforms the raw inputs f and z.

Next to this, the first hidden layers of each network are then computed as

$$H_{f,i}^{(1)} = \alpha\Big(\textstyle\sum_{j=1}^{n} w_{ij}^{(b,1)}\, f(x_j) + b_i^{(b,1)}\Big), \qquad H_{z,i}^{(1)} = \alpha\Big(w_i^{(t,1)} \cdot z + b_i^{(t,1)}\Big), \quad i = 1, \ldots, p$$

For layers $l = 1, \ldots, L-1$, interaction between the two subnetworks occurs through multiplication of their encoded states (represented by the * blocks in fig. 10):

$$Z_{f,i}^{(l)} = \alpha\left(\sum_r w_{ir}^{(b,l)}\, H_{f,r}^{(l)} + b_i^{(b,l)}\right), \qquad H_{f,i}^{(l+1)} = (1 - Z_{f,i}^{(l)})\, U_i + Z_{f,i}^{(l)}\, V_i,$$

$$Z_{z,i}^{(l)} = \alpha\left(\sum_r w_{ir}^{(t,l)}\, H_{z,r}^{(l)} + b_i^{(t,l)}\right), \qquad H_{z,i}^{(l+1)} = (1 - Z_{z,i}^{(l)})\, U_i + Z_{z,i}^{(l)}\, V_i,$$

for $i = 1, \ldots, p$. This formulation allows the trunk and branch hidden layers to exchange information at every layer through the encoded representations U_i and V_i.

At the final layer,

$$H_{f,i}^{(L)} = \alpha\Big(\sum_r w_{ir}^{(b,L)}\, H_{f,r}^{(L-1)} + b_i^{(b,L)}\Big), \qquad H_{z,i}^{(L)} = \alpha\Big(\sum_r w_{ir}^{(t,L)}\, H_{z,r}^{(L-1)} + b_i^{(t,L)}\Big)$$

The operator output is then obtained via the standard DeepONet inner product (green cross block in fig. 10):

$$\mathcal{G}(f)(z) = \sum_{i=1}^{p} H_{f,i}^{(L)}\, H_{z,i}^{(L)}$$

This causes the branch and trunk network to interact with each other before being combined linearly through the inner product. If there are non-linear relations between the trunk and branch network, this will now be picked up, improving performance (according to [61]). The architecture is illustrated in fig. 10.

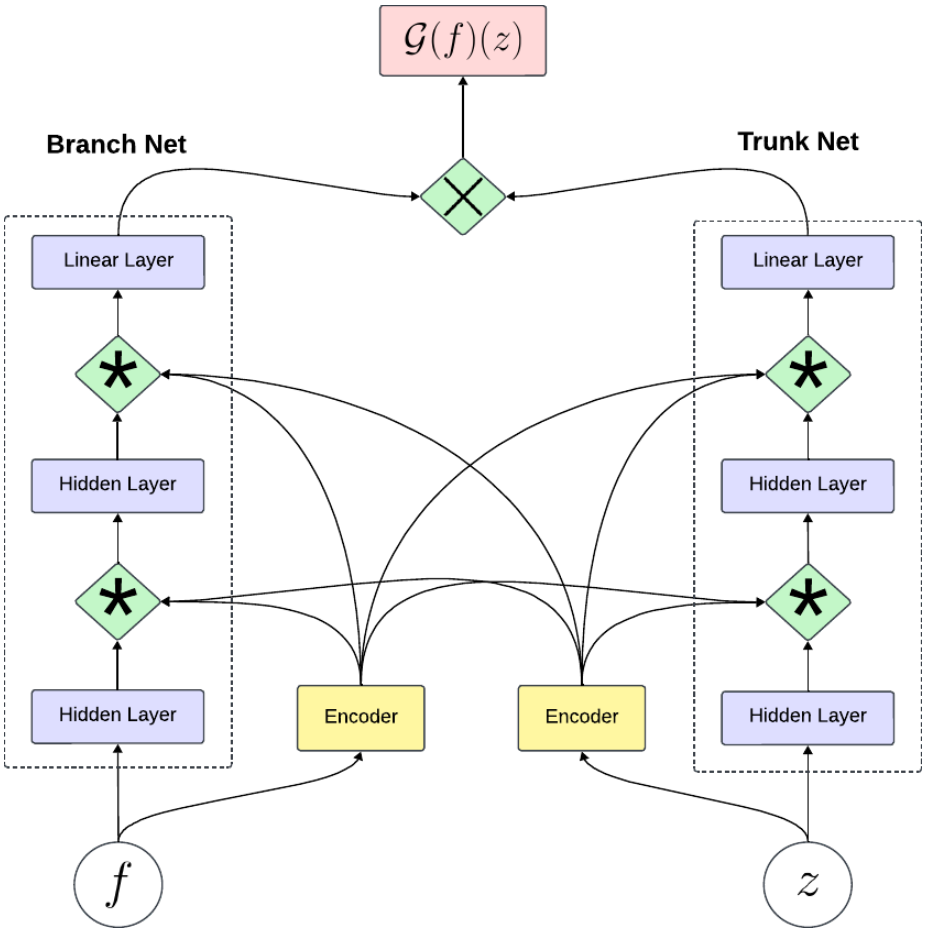


Figure 10: Intertwined architecture, the * blocks showcase the interaction with the yellow encoding blocks. The cross block at the end is the inner product that is also present in a classical DeepONet architecture. Adapted from [61].

4.2 Model architecture

We build on existing operator learning architectures to develop a new, more expressive model. The foundation of our design is inspired by the MIONet framework [21], which constructs m branch networks corresponding to m input functions. The outputs of these branch networks are then (linearly) combined with the output of a trunk network, as described in theorem 3 (fig. 9). However, to better capture the non-linear dependencies between the input functions and

the trunk input z, we extend this architecture by introducing interaction mechanisms between the branch and trunk networks. While the MIONet relies solely on linear combinations, our approach aims to model more complex relationships that such linear interactions may fail to capture.

In the intertwined DeepONet [61], the interaction between the subnetworks is implemented as a fixed weighted sum of the branch and trunk encodings (fig. 10). At each interaction layer, this mixing follows a deterministic rule of the form $(1 - Z_{f,i}^{(l)})\, U_i + Z_{f,i}^{(l)}\, V_i$, where the coefficients $(1 - Z_{f,i}^{(l)})$ and $Z_{f,i}^{(l)}$ define a convex combination that is structurally the same across all layers. So, while $Z_{f,i}^{(l)}$ may vary with the layer, the form of the weighted sum remains fixed. In our case, we need to apply this idea to multiple input functions, each representing distinct physical phenomena that can also interact (as explained in later sections). Since we have more than two encoder blocks interacting at each layer (rather than only U_i and V_i), we must determine how their weighted combination is defined. We can follow a similar procedure as in the intertwined DeepONet [61] and fix weights by beforehand deciding on a weighted sum, based on prior physical knowledge. We however, go one step further and take inspiration from transformers [58] by incorporating adaptive weights that decide the importances of each encoding at every layer.

Similar to [68, 50], we implement a softmax gating function [22] after learning a feature representation for each encoding. To manage the increasing complexity that arises when multiple input functions are present, it is not necessary for all inputs to interact at every layer. The proposed architecture therefore allows selective coupling between encoder blocks, enabling the user to determine which interactions occur between specific branch and trunk networks. This can be guided by prior physical knowledge or modeling considerations, allowing only meaningful or physically relevant dependencies to be learned. Practically, the user specifies which input functions are permitted to interact and which remain independent. When no interactions are defined between the encoders, the architecture naturally reduces to the standard MIONet.

We now consider m input functions for the m branch networks, together with

input z for the trunk network. Each branch network processes its respective input function f_j, sampled through its basis projection $\varphi_i^{(q_j)}(f_j), j = 1 \dots m$, in accordance with the construction of the MIONet (theorem 3). We define the input vector as:

$$\mathbf{c} = \begin{pmatrix} \varphi_1^{(q_1)}(f_1) \\ \varphi_2^{(q_2)}(f_2) \\ \vdots \\ \varphi_m^{(q_m)}(f_m) \\ z \end{pmatrix}, \qquad c_j = \begin{cases} \varphi_j^{(q_j)}(f_j), & j = 1, \dots, m, \\ z, & j = m+1. \end{cases}$$

Each input c_j passes through its own encoding layer of $i = 1, \dots p$ neurons with corresponding weights and biases $w_{irj}^{(e)}$ and $b_{ij}^{(e)}$ (yellow block in fig. 11). This produces the encoded features

$$E_{ij} = \alpha\left(\sum_r w_{irj}^{(e)}\, c_j(r) + b_{ij}^{(e)}\right), \qquad i = 1, \dots, p, \quad j = 1, \dots, m+1,$$

where $\alpha(\cdot)$ denotes the activation function, and $c_j(r)$ represents the r-th component of the projected input vector c_j. This corresponds to the basis coefficient or, equivalently, the sampled value of the input function at sensor location x_r for the branch networks.

Each input is also independently processed through its own first hidden layer:

$$H_{ij}^{(1)} = \alpha\left(\sum_r w_{irj}^{(1)}\, c_j(r) + b_{ij}^{(1)}\right), \qquad i = 1, \dots, p, \quad j = 1, \dots, m+1$$

At each subsequent hidden layer $l = 1, \dots, L-1$, we first compute an intermediate representation:

$$Z_{ij}^{(l)} = \alpha\left(\sum_r w_{irj}^{(l)}\, H_{rj}^{(l)} + b_{ij}^{(l)}\right), \qquad i = 1, \dots, p, \quad j = 1, \dots, m+1$$

For each input j, we define a set of interaction indices $\mathcal{I}_j \subseteq \{1, \dots, m+1\}$,

which specifies the other inputs that j interacts with at layer l. The user configures these interaction indices at the start of the model training. Attention logits are computed for every interacting pair (j, k), where $k \in \mathcal{I}_j$:

$$\beta_{jk}^{(l)} = \sum_r A_{rjk}^{(l)} Z_{rj}^{(l)} + b_{jk}^{(A,l)}, \qquad k \in \mathcal{I}_j$$

Here, $A_{rjk}^{(l)} \in \mathbb{R}$ and $b_{jk}^{(A,l)} \in \mathbb{R}$ are the trainable parameters (weights and biases) of the attention mechanism, while $\beta_{jk}^{(l)}$ denotes the resulting attention logit. The corresponding attention coefficients are obtained via a softmax over the interaction subset:

$$\alpha_{jk}^{(l)} = \frac{\exp\left(\beta_{jk}^{(l)}\right)}{\sum_{k' \in \mathcal{I}_j} \exp\left(\beta_{jk'}^{(l)}\right)}, \qquad k \in \mathcal{I}_j$$

Finally, the hidden state of each input j is updated as a weighted combination of its interactions:

$$H_{ij}^{(l+1)} = \sum_{k \in \mathcal{I}_j} \alpha_{jk}^{(l)} E_{ik}, \qquad i = 1, \ldots, p, \quad j = 1, \ldots, m+1$$

The process of computing attention logits and applying them as a weighted combination with the encodings happens in the * block in fig. 11.

The final stage retains the low-rank structure introduced in the default MIONet formulation (see theorem 3). After the attention-based interactions between subnetworks, the final hidden representations of the m branch encoders are combined through an elementwise (Hadamard) product to form the joint branch feature:

$$B_{k_1,\ldots,k_m}(f_1, \ldots, f_m) \;=\; \prod_{j=1}^{m} H_{k_j,\,j}^{(L)}, \qquad k_j = 1, \ldots, p$$

The trunk network produces its own latent features indexed by the same multi-

index $(k_1, \ldots, k_m)$,

$$T_{k_1,\ldots,k_m}(z) \;=\; H^{(L)}_{k_{m+1},\,m+1}(z),$$

and the final operator output is then obtained by summing over the shared dimensions (green cross block in fig. 11):

$$\mathcal{G}(f_1, \ldots, f_m)(z) \;=\; \sum_{k_1=1}^{p} \cdots \sum_{k_m=1}^{p} B_{k_1,\ldots,k_m}(f_1, \ldots, f_m)\, T_{k_1,\ldots,k_m}(z) \;+\; b$$

This mirrors the final combination step of the low-rank MIONet. The adaptive attention-based coupling introduced in the hidden layers generalizes the fixed multiplicative fusion used in MIONet. When these interactions are disabled, the architecture reduces to the standard low-rank MIONet.

Another option would be to fully incorporate self-attention within the variable encodings. This, however, would significantly increase the model complexity and computational cost, as additional query, key, and value projections would need to be optimized [58]. Whilst this avenue could be interesting to pursue in the future, we leave it as an extension and focus on the simpler softmax implementation.

The motivation behind introducing these interaction layers is to enable adaptive coupling between the branch and trunk subnetworks, allowing the network to model non-linear dependencies and cross-correlations between physical inputs. In the standard low-rank MIONet formulation, each branch contributes independently to the operator approximation before aggregation. By contrast, our construction allows information exchange at intermediate layers, so that features from one branch can influence another before the final combination step. This mechanism could also be incorporated into the high-rank MIONet formulation, where interactions would occur across the higher-dimensional joint feature space rather than through elementwise coupling.

We ensure that the proposed architecture remains consistent with the universal approximation theorem stated in theorem 3. Although additional interaction mechanisms are introduced, the network still satisfies the structural

requirements for universality: each subnetwork applies continuous nonpolynomial activation functions; the branch and trunk mappings remain continuous on compact subsets of their respective Banach spaces; and the operator output is obtained through a finite composition and summation of these continuous nonlinear transformations. Hence, the architectural modifications preserve the functional form required so that $\mathcal{G}$ can be expressed as a superposition of nonlinear basis functions parameterized by neural networks. Therefore, the modified model retains its theoretical capability as a universal operator approximator, capable of approximating any continuous nonlinear operator that maps between infinite-dimensional function spaces, given sufficient capacity and appropriate training. An example of a structure of the modified architecture is visible in fig. 11.

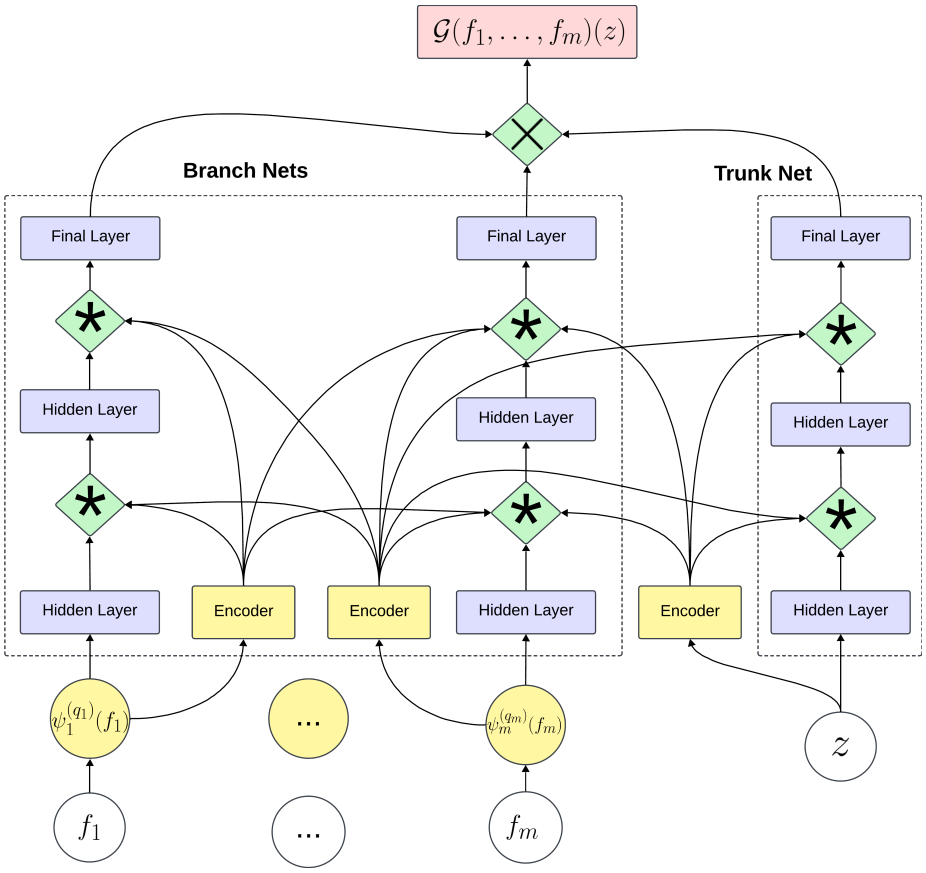


Figure 11: Modified DeepONet. In this example, the branch network from input f_1 and branch network from f_m interact, as well as f_m with the trunk network. The * blocks indicate interactions of the different inputs in between hidden layers, whereas the final cross block is the combination of the final branch and trunk features.

4.3 Local trunk

In addition to the interaction mechanism, we introduce a local trunk, which modifies how the trunk input z is processed over its evaluation domain. In the classical DeepONet, the trunk network operates in a global manner, where the entire set of evaluation points $z = (z_1, \ldots, z_W)$ is jointly encoded into a single global representation $T_{k_1,\ldots,k_m}(z)$ that is shared across all branch evaluations within the sample. This global trunk defines a common basis of functions over

the input domain and is suitable when the underlying operator exhibits coherent behavior across all z_i.

In the local trunk formulation, each evaluation point z_i is processed independently by the same trunk network, i.e., with shared parameters but without any coupling between different evaluation points. Formally, for each z_i,

$$T_{k_1,\ldots,k_m}(z_i) \;=\; H^{(L)}_{k_{m+1},\,m+1}(z_i), \qquad k_{m+1} = 1,\ldots,p,$$

This produces a set of trunk feature vectors $\{T_{k_1,\ldots,k_m}(z_i)\}_{i=1}^{W}$, each corresponding to a distinct evaluation point z_i in the input domain. Each branch network is evaluated at the same positions, yielding $H^{(L)}_{k_j,\,j}(t_i)$ for $j = 1,\ldots,m$ and $k_j = 1,\ldots,p$. The joint branch feature associated with z_i follows the same low-rank fusion introduced in theorem 3:

$$B_{k_1,\ldots,k_m}(f_1,\ldots,f_m;z_i) \;=\; \prod_{j=1}^{m} H^{(L)}_{k_j,\,j}(z_i)$$

The corresponding operator output at each local evaluation point is then

$$\mathcal{G}(f_1,\ldots,f_m)(z_i) \;=\; \sum_{k_1=1}^{p}\cdots\sum_{k_m=1}^{p} B_{k_1,\ldots,k_m}(f_1,\ldots,f_m;z_i)\,T_{k_1,\ldots,k_m}(z_i) \;+\; b$$

The local trunk processes each evaluation point independently, providing a distinct representation $T(z_i)$ at every z_i, while sharing the same network weights across all points. This contrasts with the global trunk, which produces a shared basis $T(z)$ that is reused across all evaluations. The local formulation allows the architecture to model fine-grained or spatially varying behavior of the operator, while the global formulation enforces a coherent basis shared across the entire domain. Both variants retain the low-rank fusion structure and theoretical guarantees of the MIONet formulation, differing only in how the trunk features are generated over the input domain.

The motivation for adopting this approach arises from the practical realities of dredging. Each dredging trip can differ substantially due to changing

environmental conditions and operational constraints. As a result, the behavior of onboard sensors often varies from trip to trip, even though the underlying physical principles remain consistent. These variations introduce unobservable factors that are challenging to model directly. Consequently, relying on a global trunk to capture the temporal evolution across an entire trip is insufficient, as it fails to account for localized, trip-specific dynamics. In contrast, a local trunk encodes each time step independently, enabling the network to more effectively capture short-term patterns and context-dependent variations within smaller windows (as opposed to a whole trip). The effectiveness of the local trunk is demonstrated through its superior performance compared to the global trunk, as discussed alongside the applications of the classical DeepONet and MIONet in appendix B.

At the same time, it remains necessary to capture broader temporal dependencies across the full duration of a trip. While this could be achieved by expanding the architecture to include a dedicated global encoder (alongside the local trunk), we instead propose an alternative solution: the integration of a global corrector. As discussed in section 2.4, the true value of the mixture density becomes available a few seconds after the model issues its prediction. We leverage this delayed ground truth to construct a corrector model that compensates for local biases and enhances the model's overall generalization capability.

4.4 Corrector model

On top of the modified DeepONet, we implement an online corrector mechanism. We explain this mechanism at the hand of the dredging context.

Depending on the mixture velocity, the density sensor is delayed by a certain amount of seconds. This in turn means that the true θ value is delayed by the same amount of seconds. We implement a rolling mean error correction mechanism. Let $\hat{\theta}_i$ denote the model prediction at time t_i, and θ_i the corresponding ground truth value. We assume a lag of τ seconds, meaning that at time t_i, the most recent usable ground truth is from time $t_i - \tau$.

For each prediction at time t_i, we define a correction window of length c_w (in seconds) ending at the latest available ground truth time $t_i - \tau$. The window spans:

$$[t_i - \tau - c_w,\ t_i - \tau]$$

We identify all timestamps t_j such that $t_j \in [t_i - \tau - c_w,\ t_i - \tau]$, and compute the mean error over that window:

$$\epsilon_i = \frac{1}{|\mathcal{W}_i|} \sum_{j \in \mathcal{W}_i} (\hat{\theta}_j - \theta_j)$$

where $\mathcal{W}_i = \{j \mid t_j \in [t_i - \tau - c_w,\ t_i - \tau]\}$ is the index set of valid correction points.

The corrected prediction is then given by:

$$\hat{\theta}_i^{\text{corr}} = \hat{\theta}_i - \epsilon_i$$

If no past data exists within the correction window, the prediction is left uncorrected.

This correction is applied continuously and independently for each timestamp, enabling online adjustment of predictions based on recent model performance. The implementation of the corrector will correct for the local biases learned from the modified DeepONet, and give a global correction of what is happening within a trip. That is to say, the general behavior conforming to a single trip is captured through the corrector model. The flow diagram of the correction methodology is represented in fig. 12.

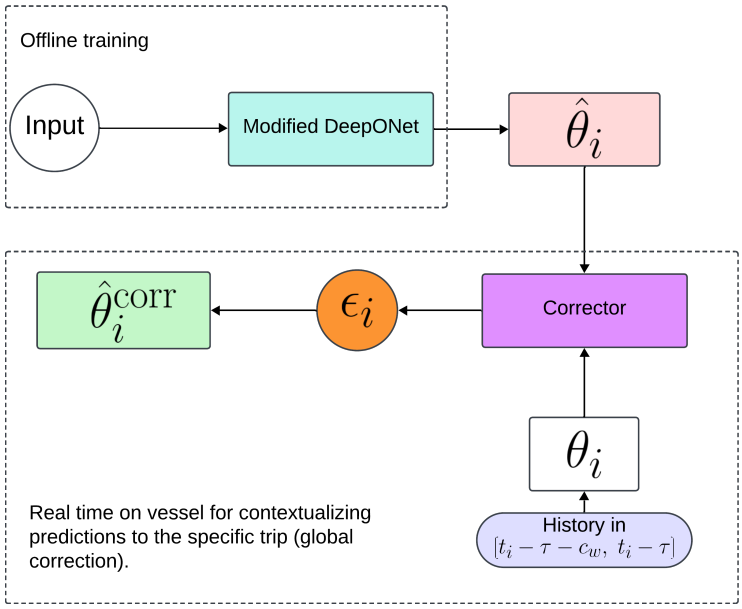


Figure 12: Flow diagram of how the corrected values are computed, we see how the online correction takes the offline predictions into account.

4.4.1 Outlier detection

To improve system reliability, we incorporate an outlier detection mechanism into the error correction pipeline. The main purpose of the correction is to adjust future predictions, but we extend the mechanism to capture anomalies within the system. These anomalies are present when there are abnormally large deviations between predicted and observed values.

At each time step t_j, we compute the prediction error $e_j = \hat{\theta}_j - \theta_j$. If the magnitude of this error exceeds a certain threshold δ, i.e.,

$$|e_j| > \delta,$$

then the corresponding data point is flagged as an outlier. The threshold δ can be chosen based on expected noise levels or system behavior. These outliers are also excluded from the correction window to prevent anomalous points from

biasing the rolling mean error:

$$\mathcal{W}_i = \left\{ j \;\middle|\; \begin{array}{l} t_j \in [t_i - \tau - c_w, t_i - \tau] \\ \text{and } |\hat{\theta}_j - \theta_j| \leq \delta \end{array} \right\}$$

This outlier rejection procedure provides a mechanism for real-time anomaly flagging. Persistent exceeding of the threshold may indicate faults in the sensors or process, serving as an outlier detection method within the system.

5 Data & Implementation

In this section, we explain how the data is structured and which variables we use to train the model. We spend some time explaining how the simulated dataset is constructed. After this, we showcase how we conform the data to the modified DeepONet described in section 4.2. Finally, we describe the procedure to train the model.

5.1 Data

The sensors aboard the vessel records a measurement every second. This is done for every TSHD that Boskalis is making use of. An example of a data excerpt is shown in table 1. A full dredging cycle usually lasts around 30 minutes, depending on the task and conditions.

Table 1: Sample excerpt of process measurements (dummy data).

Date	**2023-11-08**	**2023-11-08**	**2023-11-08**
Time	**11:02:02**	**11:02:03**	**11:02:04**
Mixture density (kg/m^3)	1220.2	1219.3	1219.8
Velocity in pipe (m/s)	2.25	2.35	2.13
Vacuum pressure (SB) (100 kPa)	-0.25	-0.25	-0.23

In table 1 we see three different variables. All of these variables have an effect on the parameter we are predicting, θ. In reality, there is a list of roughly 100 variables available in the dataset that can be selected as input functions. Since the output parameter evolves over time, the trunk network naturally represents the evaluation domain through the corresponding time steps. This allows the operator to learn mappings from temporal sensor signals to time-dependent target quantities. Furthermore, we have no spatial coordinates available, so the time coordinates are the most logical.

5.1.1 Relations

We need to have a clean and controllable data set to test the model first. Testing on real-world data from the beginning would make it difficult to determine whether observed model behavior is a result of architectural design or noise and inconsistencies in the data. Since we do not have access to a clean data set, we have to simulate it. We also do not have access to all equations relating the input functions (for the branch networks) to the output (parameter θ). So, in cohesion with knowledge from experts (A. Nobel, personal communication, March 2025), we set up relations ourselves.

Before this, we decide on the variables that we want as input functions for the branch network. We look for variables that have an effect on our loss parameter θ. In order to build the architecture of our branch and trunk networks we also have to know which (input) variables interact together. If two variables are correlated linearly, then their branch networks do not intertwine and can stay separate (since their outputs will already be combined linearly). If there is a non-linear correlation, then the parameters of the different networks are shared. We list the input variables below, together with the set up relations to the output, θ.

Change in pressure draghead

Apart from the vacuum equation, we also have an equation that represents the change in the pressure of the draghead, as mentioned in section 2. This is given as:

$$\Delta \mathrm{P}_{\mathrm{Head}} = \rho_w g(d_1 - d_2) - \frac{1}{2}\rho_m v^2(1+\alpha) - h_{head}\rho_m g$$

We remind that ρ_w is the water density, g gravity, d_1 the water depth, d_2 pump depth, ρ_m mixture density, v velocity in the pipe, α the entrance loss factor, and h_{head} the height of the draghead. We defined $\theta = (1 + \alpha + \xi + \lambda\frac{L}{D})$ for ξ the sum of losses due to curves, etc, λ the resistance factor in the pipeline

and L, D the length and diameter of the pipe, respectively. Assuming every other variable remains fixed, from the above equation we set up the relation between θ and $\Delta \mathrm{P}_{\mathrm{Head}}$ as:

$$\theta = c_{1,\Delta P_{head}} \Delta P_{head} + c_{2,\Delta P_{head}}$$

Where $c_{1,\Delta P_{head}}, c_{2,\Delta P_{head}} \in \mathbb{R}$ are constants.

Vacuum pressure

For this variable we refer to the vacuum equation (eq. (1)). That is, the vacuum pressure (V) within the pipe whilst dredging is given by:

$$V = \theta \frac{v^2}{2} \rho_m + (d_1 - d_2) \rho_m g - d_1 \rho_w g$$

So, in a controlled scenario (taking the other variables fixed), we know that:

$$\theta = c_{1,V} V + c_{2,V}$$

Again, $c_{1,V}, c_{2,V} \in \mathbb{R}$ are constants.

Mixture velocity

Similar to the vacuum pressure, we know the relation between the mixture velocity: v and θ from the vacuum equation in a controlled environment:

$$\theta = \frac{c_{1,v}}{v^2} + c_{2,v}$$

Density

For the relation between the mixture density, ρ_m and θ, we again already know it based on the vacuum equation:

$$\theta = \frac{c_{1,\rho_m}}{\rho_m} + c_{2,\rho_m}$$

Lower pipe angle

For this variable we do not have a given equation so we will have to come up with a relation ourselves. There are two sensors which fall under this umbrella of the angle. Data for the roll angle of the draghead is collected This measures the sideways angle of the draghead. The other angle is the lower pipe angle. Both of these measurements have an effect on θ. If the draghead is deeper in the soil, it causes a higher pressure loss. This means that the larger the lower pipe angle is, the more pressure is lost. This is the same for the roll angle.

We make a simple assumption and say that as the pipe is lowered, causing an increasing angle, the loss term θ follows linearly, so we have:

$$\theta = c_{1,\delta}\delta + c_{2,\delta}$$

For δ representing the lower pipe angle with constants $c_{1,\delta}, c_{2,\delta} \in \mathbb{R}$. The effect of the role is more difficult to quantify because this depends solely on where the ship is and exactly how the ground looks like. This is why we do not take the roll angle into account at this moment.

Draghead & pump depth

The relation between the draghead depth can also be found back in the vacuum equation. This is based on the formula for hydrostatic pressure [59] which states that as the draghead goes deeper below sea level, the hydrostatic pressure increases. If the pressure increases, that also means there is a higher pressure loss in the θ parameter:

$$\theta = c_{1,D}D + c_{2,D}$$

While there is a direct signal measuring the depth of the draghead, the vessels draught is also recorded, expressed in pressure units. The draught represents the vertical distance between the waterline and the lowest point of the hull, indicating how deep the vessel is submerged. By converting the pressure-

based draught signal to meters using the hydrostatic pressure relation, we can determine the vessel's submergence in the water. The limitation of the draghead depth signal is that it is referenced to chart datum, a fixed vertical reference level, rather than the water surface. By combining the draught signal with the draghead depth (relative to chart datum), we obtain the true vertical distance of the draghead below the water surface. Consequently, the D used in the equation represents the actual underwater depth of the draghead, derived from the combination of these two signals.

Jet pressure

There are jet pumps positioned at the draghead. These can be turned on to cut the soil before it enters the draghead. The higher the jet pump pressure (P_j) is, the more powerful this effect is, and so more material is able to enter the draghead. This increases the density of the mixture and in turn, decreases the pressure lost (i.e., θ). After consulting with experts (A. Nobel, personal communication, March 2025, [41]), we assume an inverse relation, given by:

$$\theta = \frac{c_{1,P_j}}{P_j} + c_{2,P_j}$$

Swellcompensator

The final variable we consider is the pressure of the swellcompensator. This refers to a mechanism on the vessel that lifts the draghead above the seabed [34]. If there is a higher pressure on the swellcompensator it means that the draghead is hovering more above the seabed. This decreases the pressure losses, i.e.,, θ. If there is less pressure on the swellcompensator (S_c), the draghead is harder on the seabed, causing an increase in θ. We assume this relation to be linear and define a relation as:

$$\theta = -c_{1,S_c} S_c + c_{2,S_c}$$

We note that all constants $c_1, c_2 \in \mathbb{R}$ represent scaling factors that deter-

mine the relative magnitude at which the corresponding variables influence one another. Since the present formulation focuses on establishing the general relationships, no additional physical constraints or assumptions are imposed on these coefficients. After we have simulated separate data for each variable, we combine them with a weighted sum to come up with our final linear model:

$$\theta_{sim} = w_1\theta_1 + w_2\theta_2 + w_3\theta_3 + w_4\theta_4 + w_5\theta_5 + w_6\theta_6 + w_7\theta_7 + w_8\theta_8$$

Where the weights signify the importances of the different variables. These weights will be determined by performing a sequential least squares optimization [14]. In fact, most of the variables are not independent of each other and this linear model does not capture all the complexities. However, since there is no other current model that can be used for simulating data, we opt for this approach.

Other variables

Some variables that haven't been mentioned yet are the draghead visor angles. Larger visor angles could lead to higher losses, after internal discussions (A. Nobel, personal communication, March 2025) this does not seem to be the most important. Furthermore, the position of the vacuum relief valve was first considered as a variable. This was later removed because the relief valve does not have a direct influence on the pressure losses. It is mainly used by the crew to change the densities of the mixture at a given moment. Another variable that was discussed is the trailing speed of the vessel. If the vessel moves faster, it would mean more soil is dredged. This was left out because it was deemed to have more of an indirect effect on θ rather than directly influencing it.

5.1.2 Variable intertwinement

Most of these variables have a direct influence on each other. The question is whether these are linear or non-linear. Based on Bernoulli's principle and the

Darcy-Weissbach equation (i.e., the vacuum equation). We know that the mixture velocity has a non-linear correlation with the vacuum pressure, change in pressure draghead, draghead depths, and the mixture density. Furthermore, we also know from the equation that the density is correlated non-linearly with both the draghead depth, as well as the change in pressure draghead. Another non-linear interaction that is present between variables is the correlation between the jet pump pressure and the change in pressure at the draghead. As the jet pumps become more powerful, there is less of a pressure loss. This correlation could look more exponential or logarithmic, depending on the soil (A. Nobel, personal communication, March 2025). A further non-linear relation could lie between the pipe angle and the velocity. If the angle is steeper, the flow might experience more resistance from gravity, this relation would be non-linear. Finally, we also need to determine which variables interact with the trunk network which in our case, represents the time steps. We decide to have the trunk network interact solely with the mixture density. The motivation behind this is that every input variable is inputted in the correct respective time step, whilst the mixture density always lags behind. This means that when building our model architecture, the variables that have a non-linear relationship will have intertwining branch networks, whereas the mixture density branch will intertwine with the trunk network; which we demonstrate in section 5.2.

5.1.3 Simulated dataset

For the aforementioned equations, we run inference on real data. As in, we find the values of the scaling factors $c_1, c_2 \ldots$ by testing the set up relations on real data. We do this by generating θ values via the vacuum equation. Using these values we find the best fit for every relation. We discuss how well the relation fits to the process. This way, we can check whether the relations hold. After the values of the scaling factors are determined, we create the simulated dataset for θ from the relations and sensor inputs (of the vacuum, mixture density, etc.). Whilst this may be a good check, it is not the most accurate

because a change in θ can be caused by a different variable that is not being incorporated into a relation.

We take 150,000 data points when the vessel "Strandway" is dredging. From these data points we calculate θ values based on a rolling window approach. After this, we filter out values of θ that are outliers ($0 \leq \theta \leq 5$). This is about 20% of the data points. Then we fit the relations one by one. The details of the procedure together with explanations can be seen in appendix A. We provide the final equations to simulate data below:

$$
\begin{aligned}
\theta_{\Delta P_{Head}} &= 0.80\Delta P_{head} + 2.00 \\
\theta_{Vac} &= 2.00 - 1.50V \\
\theta_{Density} &= \frac{1400}{\rho_m} + 1.50 \\
\theta_{Vel} &= \frac{4.00}{v^2} + 2.00 \\
\theta_{Angle} &= 0.02\delta + 2.50 \\
\theta_{Jet\ pumps} &= \frac{2.50}{P_j} + 2.30 \\
\theta_{S_c} &= -0.04S_c + 4.50 \\
\theta_{DHDepth} &= 0.05D + 2.00 \\
\theta_{Sim} = 0.28\theta_{\Delta P_{Head}} + 0.18\theta_{Vac} + 0.13\theta_{S_c} + 0.12\theta_{Vel} + & \\
0.11\theta_{Density} + 0.08\theta_{DHDepth} + 0.05\theta_{Jet\ pumps} + 0.05\theta_{Angle} & \quad (6)
\end{aligned}
$$

We generate data using this system of equations. There are different ways we can do this. One of the options would be to sample random points of data for all the input sensors, then use these random points to generate values for θ_{sim}. The problem with this would be that it will not retain any structure of a dredging trip, so the results will be difficult to analyse. Using the real sensor data also causes issues because of how noisy & unpredictable it is, and the whole point of simulating is that we are in control of the dataset. What we decide to

do is filtering real life data.

The initial step in the filtering process involves applying a rolling average correction mechanism for the density variable. This enables the alignment of the mixture density signal with the correct time index, accounting for the time it takes for the mixture to travel through the pipe.

To match the lagged density signal with the appropriate physical context, a dynamic rolling window is computed based on the current mixture velocity. Specifically, the window size approximates the time required for the slurry to travel the pipe length. This time varies with velocity and is therefore computed individually for each timestep. This dynamic smoothing enables a physically-informed correction of the density signal.

Let $\rho_m(t)$ denote the measured mixture density at time t, $v(t)$ the mixture velocity at time t, and L the length of the pipe. The travel time of the slurry through the pipe is then given by:

$$T_{\text{travel}}(t) = \frac{L}{v(t)}$$

For a time series sampled at intervals Δt, this travel time corresponds to a discrete window length (in samples)

$$w_i = \frac{T_{\text{travel}}(t_i)}{\Delta t} = \frac{L}{v_i \, \Delta t},$$

which is bounded between $w_{\min}$ and $w_{\max}$ to avoid extreme values:

$$w_i = \max\big(w_{\min}, \min(w_{\max}, w_i)\big)$$

The density signal is then smoothed over the last w_i samples:

$$\tilde{\rho}_m(t_i) = \frac{1}{w_i} \sum_{k=0}^{w_i - 1} \rho_m(t_i - k \, \Delta t),$$

which represents an average over the time window that the slurry takes to travel

through the pipe. The smoothed signal is subsequently shifted forward by the same travel time:

$$\rho_{m,\mathrm{corr}}(t_i) = \tilde{\rho}_m(t_i + T_{\mathrm{travel}}(t_i))$$

This correction ensures that the density measurement at time t_i represents the mixture that entered the pipe approximately $T_{\mathrm{travel}}(t_i)$ seconds earlier, i.e., the same physical section of slurry that now reaches the sensor.

Optionally, a fixed time delay T_{lag} can be added to simulate this effect consistently:

$$\rho_{m,\mathrm{lag}}(t_i) = \rho_{\mathrm{corr}}(t_i - T_{\mathrm{lag}})$$

This procedure introduces velocity-dependent alignment and smoothing of the density signal, resulting in a corrected density $\rho_{m,\mathrm{corr}}(t)$. It is essential for enabling physically consistent simulation and lag-aware model training. By correcting the density signal, all measurements can be properly aligned in time, ensuring that the simulated system behaviour reflects the true physical relationships. Additionally, the introduction of fixed delays allows us to train and evaluate models under controlled lag conditions, making it possible to study the effect of density signal delays on model performance.

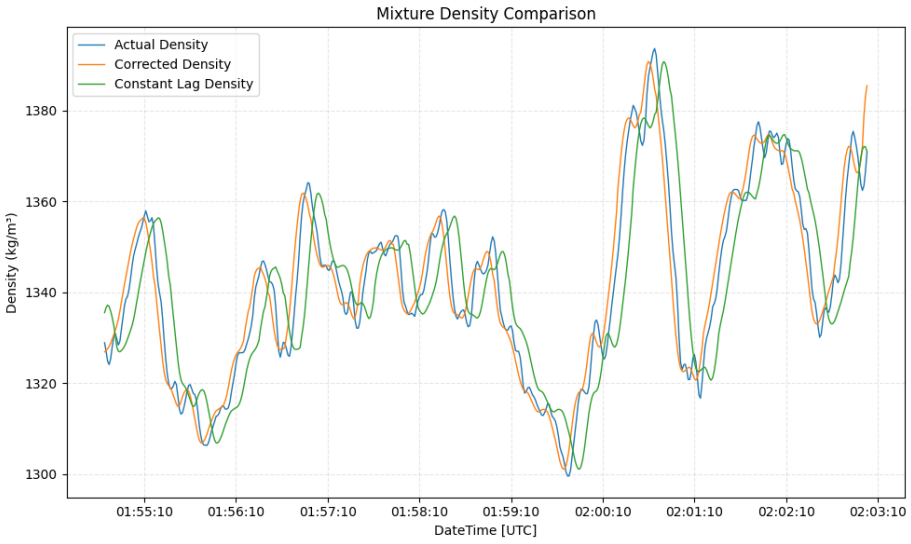


Figure 13: Comparison of actual, corrected, and constant lagged (ten seconds) density. The corrected and constant lagged density are smoothed with a dynamic rolling window.

After preprocessing the raw signals, we apply a Savitzky-Golay filter [49] on all relevant input variables. This type of filtering is suitable in physical systems like dredging, where sensor data often contains high-frequency noise but where preserving the overall trend is critical. A previous study has demonstrated its effectiveness for similar tasks in dredging applications [29].

The Savitzky-Golay filter performs local polynomial regression within a moving window to smooth the signal. For a window of size $2m+1$ over n data points, the method fits a polynomial of degree d in the form:

$$P_d(i) = \sum_{k=0}^{d} b_k i^k,$$

where b_k are the polynomial coefficients and i referring to the position of a sample within that specific window. These coefficients are obtained by minimizing the squared error:

$$E = \sum_{i=-m}^{m} \left(P_d(i) - y(i)\right)^2,$$

with $y(i)$ representing the observed values within the window. This local optimization ensures that the filter retains the underlying signal characteristics while suppressing noise.

The final step, following the smoothing procedure, involves resampling each trip to a fixed temporal resolution using linear interpolation [26].

Let a single trip provide a sequence of measurements $\{y_j\}_{j=0}^{n-1}$ for a given variable (after any prior smoothing). We reparameterize the sample index onto the unit interval via

$$x_j \;=\; \frac{j}{n-1}, \qquad j = 0, 1, \ldots, n-1,$$

and define a fixed target grid of length N,

$$x_k^\star \;=\; \frac{k}{N-1}, \qquad k = 0, 1, \ldots, N-1,$$

with N = `target_duration`. The piecewise–linear interpolant $L(x)$ through the data $\{(x_j, y_j)\}$ is

$$L(x) \;=\; y_j \;+\; (y_{j+1} - y_j)\,\frac{x - x_j}{x_{j+1} - x_j}, \qquad x \in [x_j, x_{j+1}],\; j = 0, \ldots, n-2.$$

The resampled (simulated) series $\{\tilde{y}_k\}_{k=0}^{N-1}$ is then obtained by

$$\tilde{y}_k \;=\; L\big(x_k^\star\big), \qquad k = 0, 1, \ldots, N-1$$

Both the original and target time axes are normalized to the unit interval $[0, 1]$. Hence, if a trip originally contains more or fewer samples than the target duration, the interpolation implicitly stretches or compresses the signal in time such that all trips share the same number of uniformly spaced points N.

This ensures that all trips share a consistent duration and number of samples, facilitating uniform input dimensions for both model training and subsequent analysis. Standardizing the temporal grid also simplifies the comparison of predicted and actual signals across trips of varying lengths. Only trips that meet

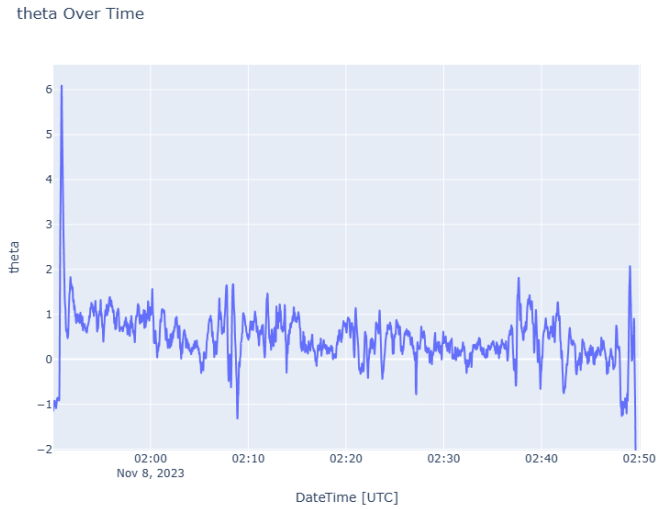
predefined criteria for duration and data quality are retained for further processing. After the input variables are prepared in this manner, the corresponding target values for θ_{sim} are computed by simulating the physical system described in eq. (6).

We emphasize once more the importance that the model is trained on data where no sensor outliers are present. Previous research has been done (O. van de Ven, personal communication, November 2024) to discover time periods when there were no sensor outliers. The period has been identified from November 8, 2023 up to November 20, 2023. We apply the filtering process for this time period.

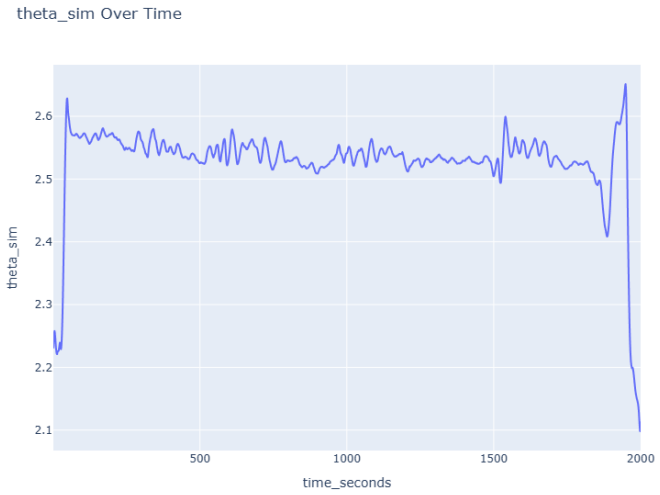
Regarding the specifics, we chose a minimum and maximum window size of $w_{min} = 1, w_{max} = 50$ for the dynamic smoothing. A window size of 50 time steps with a polynomial of order three was chosen for the Savitsky-Golay filtering. Finally, we opted to go for a uniform trip size of $2,000$ seconds. The upper limit of $w_{max} = 50$ corresponds to the maximum expected time of the slurry within the pipe at (especially) lower flow velocities. This ensures that the moving average does not extend beyond the physically relevant time horizon, while still providing sufficient smoothing to mitigate fluctuations. For the Savitzky–Golay filtering, the chosen window length aligns with the physical interpretation. A third–order polynomial was found to offer an appropriate trade–off between smoothness and flexibility: it captures the gradual trends of the process without introducing excessive curvature or overfitting to noise. The trip duration of $2,000$ seconds reflects the typical operational length of a dredging cycle, which usually lasts around 30-35 minutes. Applying this process resulted with a uniformly sized and denoised dataset of 56 trips, each aligned in both duration and structure.

Whilst the system appears to be accurate, its performance does not align perfectly with the real θ (see fig. 14). However, it is important to note that the correctness of the 'real' θ cannot be guaranteed, as it was simulated using an equation that neglects several variables known to influence its' behaviour. This evident by fig. 14a where we can see that θ exceeds the bounds $\theta \in [0, 5]$ given

by experts. Moreover, there exists no direct measurement or alternative method to determine the true value of θ, making it difficult to perform a reliable comparison. What can be stated about the simulated data is that it is controllable and provides a reasonable representation of the behaviour of θ under idealized conditions.



(a) One entire trips worth of data of actual θ. We can see how it over and under shoots the bounds $\theta \in [0, 5]$, whilst also fluctuating alot.



(b) The corresponding simulated θ values from the trip in fig. 14a. The behaviour is less erratic and does not exceed the bounds, whilst still maintaining similar characteristics to the original trip.

Figure 14: Comparison of a trip with real θ values and the simulated trip based off of it.

5.2 Implementation

Here we discuss how we apply the operator learning methodology from section 4.2 to our data. We also go through the training procedure and the methodology we use to evaluate the performance of the model.

5.2.1 Architecture

The architecture learns a mapping between two function spaces: the sensor values to the loss parameter. We also need to keep in mind what the goal for the model: we want to get close to real-time predictions of the loss parameter, whilst the vessel is still dredging. This means that training a model on a full trip worth of sensor data to get a predicted output for θ on that full trip is not of much use to us. In real-time, this would mean that we see the development of θ for a whole trip, only after that trip is done (since we need the sensor inputs of that whole trip to make a prediction). The aim of the project is for the crew aboard to react to the θ value in order to optimize suction production. This is why we will train on windows with size w within a trip instead of on a whole trip, the details of which are explained in section 5.2.2.

The input functions f_i to the branch network represent the different sensor variables that influence the loss parameter θ. Each function corresponds to one variable, evaluated over a temporal window of length w, such that every window

constitutes one training segment. For the k-th window, we define:

$$
\begin{aligned}
f_1(t_{s_k}, \ldots, t_{e_k}) &= P_j(t_{s_k}, \ldots, t_{e_k}) \\
f_2(t_{s_k}, \ldots, t_{e_k}) &= \Delta P_{\text{head}}(t_{s_k}, \ldots, t_{e_k}) \\
f_3(t_{s_k}, \ldots, t_{e_k}) &= V(t_{s_k}, \ldots, t_{e_k}) \\
f_4(t_{s_k}, \ldots, t_{e_k}) &= v(t_{s_k}, \ldots, t_{e_k}) \\
f_5(t_{s_k}, \ldots, t_{e_k}) &= D(t_{s_k}, \ldots, t_{e_k}) \\
f_6(t_{s_k}, \ldots, t_{e_k}) &= \delta(t_{s_k}, \ldots, t_{e_k}) \\
f_7(t_{s_k}, \ldots, t_{e_k}) &= S_c(t_{s_k}, \ldots, t_{e_k}) \\
f_8(t_{s_k}, \ldots, t_{e_k}) &= \rho_{m,\text{lag}}(t_{s_k}, \ldots, t_{e_k})
\end{aligned}
$$

Here, t_{s_k} and t_{e_k} denote the start and end times of the k-th window, respectively, with $t_{e_k} = t_{s_k} + w$. We set the index $i = 1, \ldots, 8$ to the different input variables, while k indexes the temporal window within a trip. Each $f_i(t_{s_k}, \ldots, t_{e_k})$ therefore represents the evolution of the i-th sensor variable over the k-th time window of length w. The trunk network is defined through these windows:

$$
z = (t_{s_k}, \ldots, t_{e_k})
$$

With the branch interactions defined in table 2 as:

Table 2: Branch interaction configuration using input functions f_i. Each branch may interact with other branches and, optionally, the trunk. Interacting functions share attention logits between every layer, as was discussed in section 4.2.

Function	**Interacting Branches**	**Trunk Interaction**
f_1(Jet pumps)	$\{f_1, f_2\}$	No
$f_2(\Delta P_{\text{Head}})$	$\{f_1, f_2, f_4, f_8\}$	No
f_3(Vacuum pressure)	$\{f_3, f_4\}$	No
f_4(Velocity)	$\{f_2, f_3, f_4, f_5, f_6, f_8\}$	No
f_5(Draghead depth)	$\{f_4, f_5, f_8\}$	No
f_6(Angle)	$\{f_4, f_6\}$	No
f_7(Swellcompensator)	$\{f_7\}$	No
f_8(Lagged density)	$\{f_2, f_4, f_5, f_8\}$	Yes
Trunk	$\{f_8\}$	—

Then, the operator as described in section 4.2 is denoted as:

$$\mathcal{G}(f_1, \ldots, f_m)(z) = \mathcal{G}(f_1, \ldots, f_8)(t_{s_k}, \ldots t_{e_k}) \in \mathbb{R}$$

5.2.2 Training

As the model is trained on windows within each trip rather than on the full trip, we adopt a windowing strategy inspired by convolutional neural networks [28]. This approach is closely related to sliding-window techniques widely used in time series modeling [27, 10]. This enforces the architecture to learn operators over temporal windows of size w. We aim to learn an operator $\mathcal{G}(f_1, \ldots, f_8)(t_{s_k}, \ldots, t_{e_k})$, where each f_i represents one input variable evaluated over the k-th window. For a complete trip belonging to sensor input j with n time steps, the full signal can be expressed as

$$f_j(t_1, \ldots, t_n) = [x_1, \ldots, x_n]$$

This sequence is then segmented into windows of size w. The buffer size (commonly referred to as the stride) determines the number of time steps by which the window moves forward after each segment. Each window k for a given trip is defined as

$$f_j^{(k)} = [x_{s_k}, x_{s_k+1}, \ldots, x_{s_k+w-1}],$$

where $s_k = (k-1)s$ denotes the start index of the k-th window and s is the stride length. This procedure is repeated for all trips in the dataset. The total number of training samples per sensor input is therefore

$$T \left\lfloor \frac{n-w}{s} + 1 \right\rfloor,$$

where T denotes the total number of trips. The model is then trained to fit its parameters across all generated windows.

The loss function that is being used to evaluate every window to its respective predictions is the mean squared error. We configure the activation function in every hidden layer to be the hyperbolic tangent activation function. The optimizer of choice is adam [24].

There is no optimal choice for window size and stride when segmenting the time-series data; instead, these parameters must be selected with consideration for the real-world application. In practice, we avoid long strides, as this would reduce the frequency of predictions which is undesirable for having timely updates. Given that the density sensor typically exhibits a delay of approximately ten seconds, we set the stride (or buffer) to ten seconds, allowing the model to produce a prediction per ten-second interval. This means that the model outputs a prediction for the past ten seconds, every ten seconds. A shorter stride would lead to a much higher computational cost per trip, without necessarily improving model performance.

For the window size, we consider the range over which variables are expected to influence each other. A window that is too long risks incorporating outdated information that no longer contributes meaningfully to the model's prediction

at the current time step. In the end we decide for a window size of 100 seconds. Regarding application, this choice implies that the first valid predictions from the operator will become available after the initial 100 seconds of dredging, once the first complete window has been observed. Given a stride of ten seconds, subsequent predictions are then generated every ten seconds as new data arrive. This configuration allows for near real-time monitoring of the dredging process, with continuously updated estimates of the loss parameter θ as the operation progresses.

To train and evaluate the model, we randomly pick 70% of trips for training and 30% for testing. This results in 7,449 windows of length 100 for training, and 3,247 for testing. During training, windows are randomly batched and passed through the network; one epoch corresponds to a complete pass over all 7,449 training windows.

We note that since we constructed the modified DeepONet in such a way that it encodes the time steps explicitly via the local trunk methodology, overlapping windows will always give the same predictions. For example, the prediction for $t = 1000s$ with the same input variables will always output the same prediction for θ with the local trunk methodology. This is not the case for the classical DeepONet and MIONet however. Since we want to compare performances, we define a methodology to evaluate on overlapping timesteps in different windows. We define $\hat{\theta}_t^{(i)}$ as the prediction made for time step t by window i. Furthermore, $\mathcal{W}(t)$ represents the set of all window indices i that include the time step t.

The first strategy is averaging, where for every time step, we take the mean over all the overlapping windows that cover it:

$$\hat{\theta}_t = \frac{1}{\mathcal{W}(t)} \sum_{i \in \mathcal{W}(t)} \hat{\theta}_t^{(i)}$$

In real-time however, this is not possible since not all overlapping windows are available at the time point we want. Instead, we select the prediction from the first window that covers time step t:

$$\hat{\theta}_t = \hat{\theta}_t^{(i^*)}, \text{ where } i^* = \min\{i \in \mathcal{W}(t)\}$$

This way, we avoid waiting for future windows when getting a real-time prediction.

5.2.3 Evaluation metrics

Similar to other literature using time series forecasting with deep learning, we test our model with three metrics [52]: root mean square error (RMSE), mean absolute error (MAE), and the coefficient of determination (R^2). We note that we do not perform evaluation on the training windows, but that we aggregate all predictions back into their respective trip, and evaluate on a trip basis. Since we are evaluating on multiple trips, the metrics are then averaged to determine the overall performance.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(\hat{\theta}_i - \theta_i)^2}{\sum_{i=1}^{n}(\theta_i - \bar{\theta})^2}$$

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\hat{\theta}_i - \theta_i\right)^2}$$

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}\left|\hat{\theta}_i - \theta_i\right|$$

A smaller RMSE and MAE indicate a smaller error, whilst an R^2 value closer to 1 indicates a higher forecasting accuracy.

We analyse the performance of the modified DeepONet first, after which we see how much the corrector model improves the predictions. We compare metrics with the classical DeepONet, as well as with the MIONet.

Finally, we also use a baseline to compare our results to. The process of estimating θ is currently not being done at all, so we define a simple baseline ourselves. For this we define our baseline θ_t, as the rolling mean of its' previous w (window size) values. That is, given the ground truth sequence $\theta^{(i)} =$

$\left[\theta_1^{(i)}, \theta_2^{(i)}, \ldots, \theta_n^{(i)}\right]$ and corresponding timestamps $t^{(i)} = \left[t_1^{(i)}, t_2^{(i)}, \ldots, t_n^{(i)}\right]$ for each trip $i = 1, \ldots, T$, the rolling mean baseline $\hat{\theta}_t^{(i)}$ at time t is defined as the mean of all past values of $\theta^{(i)}$ within a rolling window of size w seconds ending at t:

$$\hat{\theta}_t^{(i)} = \frac{1}{|S_t^{(i)}|} \sum_{j \in S_t^{(i)}} \theta_j^{(i)},$$

where $S_t^{(i)} = \left\{ j \mid t_j^{(i)} \in [t_t^{(i)} - w,\, t_t^{(i)}] \right\}$ is the set of indices within the rolling window at time t for trip i.

If no valid indices exist in the window, the prediction defaults to the mean of all previous values (to account for the first w values within a trip):

$$\hat{\theta}_t^{(i)} = \frac{1}{t} \sum_{j=1}^{t} \theta_j^{(i)}$$

Once again, we need to define the length of this window. Since we split the trips into windows of length 100 seconds for our training data, we will also take the rolling window in our baseline as 100 seconds.

5.2.4 Hyperparameter tuning

There are a number of hyperparameters in our model architecture that need to be optimized. The arguably most important hyperparameter is the learning rate. This decides the steepness of the correction when doing backpropagation, as derived from the gradient descent discussions in section 4.1.1. A large learning rate could cause the update step to overshoot the optimum, whilst a low learning rate leads to a slow rate of convergence [17]. The batch size is another important hyperparameter: this determines how much of the data is seen in one training step. A batch size of one corresponds to one training window being seen in every step. The choice of batch size reflects a trade-off between speed and generalization. Large batches accelerate training but can lead to poorer generalization, while small batches enhance accuracy at the cost

of increased training times. The final hyperparameters tuned are the number of hidden layers and the size of each layer. Increasing the number of layers results in a deeper network, while increasing the layer size produces a wider model. In both cases, the total number of trainable parameters grows, leading to a more complex model to optimize. This might improve the performance of the model but can also cause gradients to vanish, making the training unstable [17]. In our case there are three types of hidden layers: the encoding layer, the first hidden layer, and the subsequent hidden layers. We differentiate between the first hidden layer and the others because there is no attention mechanism yet applied to this layer, so it serves a different purpose. The rest of the layers all serve a similar purpose (other than the encoding layer). Even though these are different types of hidden layers, we do not test configurations where they are different sizes. The hyperparameter optimization would then take too long (as is explained in the subsequent discussion). We also do not test for different sizes and number of layers in different branch and trunk networks, since this would also be too computationally expensive. We take the same structure for each branch network, with that structure resonating with the trunk network.

The way we test the performances of the different configurations is through K-fold cross validation [54]. K-fold cross-validation is a widely used evaluation technique that aims to assess the generalization ability of a model. Instead of evaluating performance on a test set, the data is divided into K equally sized folds. The model is trained and validated K times, each time using $K-1$ folds for training and the remaining fold for validation. The average loss per fold serves as an estimate of the model's expected performance on unseen data. K-fold cross-validation helps reduce the variance associated with a single train/test split, and provides a more stable and reliable measure of model performance.

The hyperparameter grid explored in this study is detailed in table 3, comprising a total of 81 unique configurations. Each configuration is evaluated using a 5-fold cross-validation on 20% of the available dataset. In four of the five folds, the model is trained on 1,719 windows (each of length 100) and evaluated on 382 windows. The remaining fold involves training on 1,528 windows and testing on

573, ensuring full utilization of the 20% data split.

Each fold for every configuration is trained for ten epochs. Although this represents a short training duration, most performance improvements are observed within these initial epochs (as will be discussed in section 6). Extending the training duration, or expanding the training set was considered but deemed computationally excessive, since each configuration already requires approximately 40 minutes to train for ten epochs. This computational cost is largely attributed to the high model complexity, with approximately 60,000 trainable parameters for an architecture consisting of five (subsequent) hidden layers of size 32 and a batch size of 32. For deeper or wider networks, this number increases substantially, further amplifying the computational demands.

For each fold and configuration, we compute the RMSE, MAE, and R^2 values. These metrics are evaluated twice: once using the predictions of the modified DeepONet model, and once after applying the corrector model. To assess overall performance, we report the average of each evaluation metric across the five folds.

As the optimal correction window for the corrector model is not yet known, a fixed correction window size of ten seconds is applied consistently across all folds. The correction window size is optimized post-training, as detailed in section 5.2.5, to again reduce the computational cost associated with model optimization.

Table 3: Hyperparameter grid used for model tuning

Hyperparameter	**Values**
Hidden layer sizes	32, 64, 128
Number of (subsequent) hidden layers	3, 5, 7
Batch size	32, 64, 128
Learning rate (LR)	0.01, 0.001, 0.0001

The hyperparameter optimization process was executed over a total of 54

hours on a single-GPU machine. Given this already substantial computational investment, we chose not to explore more granular architectural variations, such as layer-specific size differences or alternative branch/trunk designs.

In table 4, the five configurations that achieved the lowest validation losses are presented. Among the evaluated metrics, we consider the corrected RMSE to be the most relevant, as it most closely reflects the model's performance in real-world applications. Accordingly, we rank the hyperparameter configurations based on this corrected RMSE.

As shown in table 4, the top-performing configurations consistently yield the best scores across all metrics, both before and after applying the corrector model.

Table 4: Performance metrics for each hyperparameter configuration. Each cell shows the corrected value followed by the original value in parentheses.

Hidden layer sizes	**# Layers**	**Batch size**	**LR**	**RMSE** (corr. / orig.)	**MAE** (corr. / orig.)	**R^2** (corr. / orig.)
32	5	32	0.010	0.054 (0.070)	0.021 (0.048)	0.201 (-0.349)
64	7	32	0.010	0.075 (0.098)	0.034 (0.065)	-0.632 (-1.977)
32	7	32	0.010	0.075 (0.154)	0.029 (0.116)	-1.076 (-11.231)
32	3	32	0.010	0.083 (0.492)	0.035 (0.473)	-1.520 (-275.347)
32	3	64	0.010	0.089 (0.106)	0.041 (0.060)	-1.391 (-2.686)

The results presented in table 4 illustrate the impact of various hyperparameter configurations on model performance. Notably, the configuration with the smallest hidden layer size (32), a moderate network depth (5 layers), and the smallest batch size (32) achieved the best performance across all evaluation metrics. Specifically, it yielded the lowest corrected RMSE (0.054) and MAE (0.021), alongside the highest corrected R^2 score (0.201). This indicates that smaller, more compact network architectures generalize better in this context, potentially due to reduced complexity.

While all configurations shared the same learning rate (0.01) in this table, its strong performance across architectures suggests that this value may be optimal

for this setup. However, since this learning rate was the highest out of the ones tested, it implies that slightly increasing the learning rate could be a promising direction for further tuning. Another option that could be explored in future avenues would be to implement an adaptive learning rate [70], which alters the learning rate based on first order (gradient) information.

Another important observation is the consistent performance improvement across all configurations when applying the corrector model. This post-processing module improved all metrics relative to their original values. This includes substantial improvements in the R^2 score, in some cases turning negative values into positive scores. This underlines the corrector model's effectiveness in aligning predictions and correcting systematic biases (either from the predictions, sensors, or other causations).

Overall, the R^2 scores are modest, with the uncorrected outputs performing poorly: only the best configuration attain decent R^2 values. For reference, the worst performing configuration with five hidden layers of width 128, learning rate 0.01, and batch size 32 exhibited catastrophic divergence in one fold. Its fold-averaged metrics were RMSE = 24,050 (uncorrected) and 3,302 (corrected), MAE = 23,638 (uncorrected) and 1,706 (corrected), and $R^2 = -7.7 \times 10^{11}$ (uncorrected) and -1.1×10^{10} (corrected), indicating severe instability at this depth/width with a high learning rate. Importantly, the generally weak R^2 is not driven by such outliers. A contributing factor is limited data diversity: the dataset comprises 56 trips, of which 20% (roughly 11 trips) are used, and each configuration is trained for only ten epochs. This combination constrains the model's ability to generalize. In the full-model setting; where more data and training are available, these issues do not arise (see section 6).

Based on these findings, the best-performing configuration was selected to train a final model on the full dataset. For this training, we use a standard 70-30 train-test split and train the model for 100 epochs.

5.2.5 Corrector model

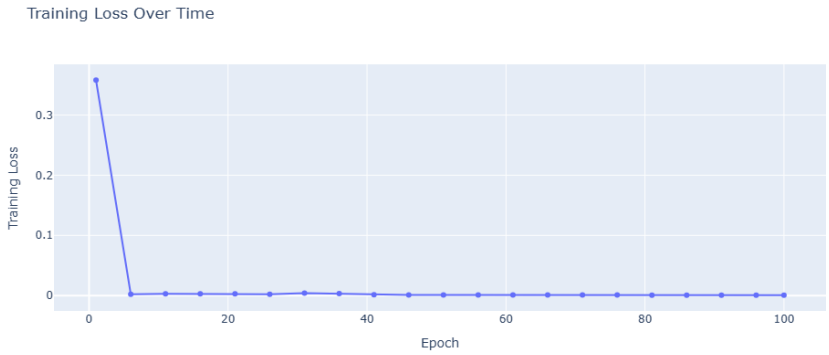
After we train the model with the modified DeepONet architecture, we implement the corrector model process that was discussed in section 4.4. This is done after the main architecture has been trained, in a post-processing context. For this post-processing we also run a short hyperparameter tuning. We need to decide how far back the rolling mean error should be looked at. After the full model has been trained we test the corrector model on correction windows for $c_w \in [1, 50], c_w \in \mathbb{Z}$, where the window is always lagging $\tau = 10$ seconds behind (constant density lag). This is tested on validation (test) data. The correction window yielding the lowest average root mean squared error per trip is then chosen, the results of which are in section 6.2. This correction window will then also be applied in all future iterations that use the corrector model. In future research, the optimal correction window can be established alongside the hyperparameter tuning of the modified DeepONet. Since we are working with controlled, simulated data, we make sure there are no outliers present in the preprocessing already. Hence, we do not showcase the additional outlier detection capabilities that the corrector model possesses in our results.

6 Results

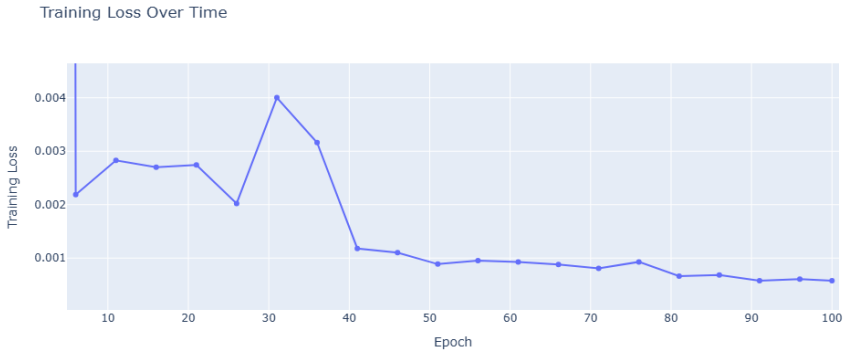
We evaluate the performance of the proposed model on both training and testing trips, and benchmark these results against the classical DeepONet and MIONet architectures, as well as to the baseline. In addition, a sensitivity analysis is performed to evaluate the robustness of the model when exposed to varying temporal timespans and when specific input features are omitted. The final part of this section explores the performance of the model on real data.

6.1 Modified DeepONet performance

The network is trained using the optimized hyperparameter configuration for a total of 100 epochs , with training taking approximately 8 hours on a single-GPU machine (NVIDIA T4) to configure 63,523 parameters. The best-performing model on the training set was obtained at epoch 78. The loss curve of the training process is detailed in fig. 15.



(a) Loss curve of the full training. The biggest decrease in the training loss already comes after the first ten epochs.



(b) The same loss curve but zoomed into the later epochs. We can see how the training loss still gradually decreases by a small amount as the epochs increase. There is also a spike in loss around epoch 30 which may have been caused by momentary instability in the gradient updates. The loss quickly stabilizes afterward, suggesting that the model recovered and continued to converge as expected.

Figure 15: Loss curves associated with the training. The training loss is logged every five epochs.

Our evaluation begins with an assessment of the model's predictive performance without incorporating the corrector mechanism. This serves as a basis for understanding the effectiveness of the core architecture. We then introduce the corrector component and analyse the extent to which it improves prediction accuracy and generalization. A comprehensive table summarizing all evaluation

metrics is provided in table 5.

To assess model performance, we present results on two representative trips from both the training and testing sets: one characterized by relatively smooth dynamics, making it easier to predict, and another exhibiting more variability and fluctuations, posing a greater challenge to the model. This is calculated by the total variation of the trip. According to [25], the total variation of a function θ defined on an interval $[a, b]$ is given by

$$V_a^b \theta = \sup \sum_{i=1}^{n} |\theta(t_i) - \theta(t_{i-1})|,$$

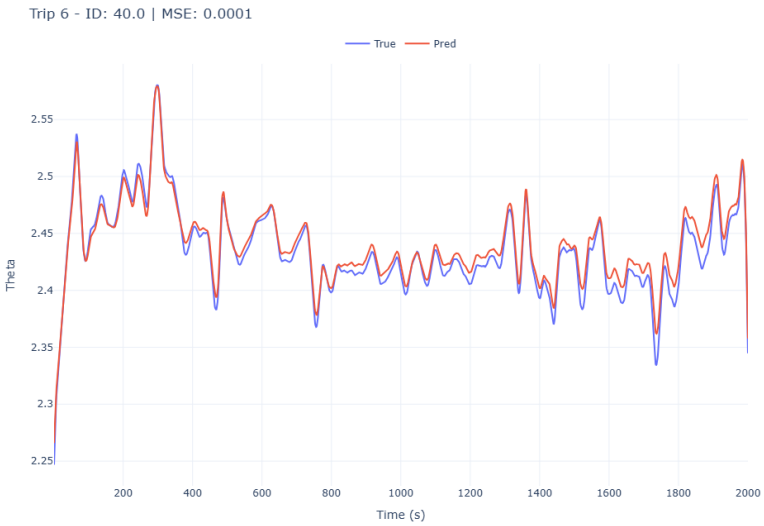
where the supremum is taken over all partitions $a = t_0 < t_1 < \cdots < t_n = b$. For discrete time series data, this can be approximated as

$$V(\theta) = \sum_{t=2}^{T} |\theta_t - \theta_{t-1}|$$

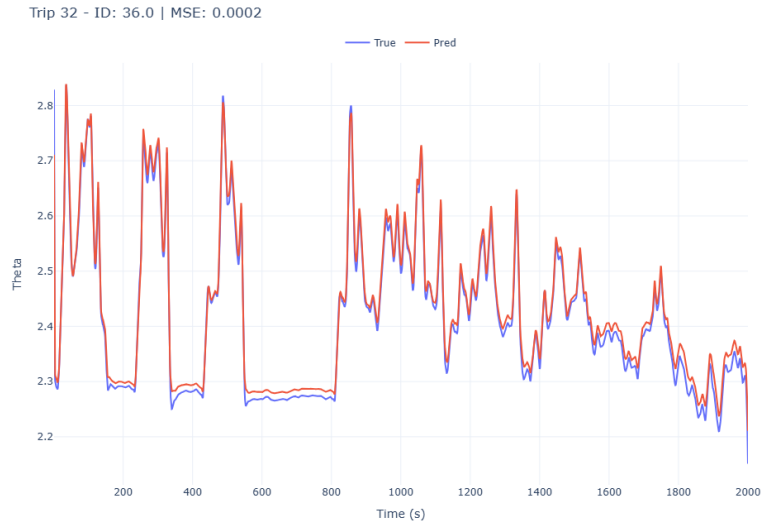
A lower total variation indicates smoother temporal dynamics, whereas higher values correspond to stronger fluctuations and increased variability. For both the training and the testing trips, we showcase two trips with differing amount of total variation. The average total variation over the 56 trips is 5.84. The total variation for every trip in the dataset is listed in appendix C.

6.1.1 Training set

On the training data, the model demonstrates strong overall performance. Nevertheless, it exhibits occasional difficulty in capturing abrupt transitions, leading to slight overpredictions in regions with rapid changes as seen in fig. 16.



(a) Predictions for a stable trip ($V(\theta) = 3.24$). We see that the prediction overshoots at some points.



(b) Predictions for the trip with more fluctuations ($V(\theta) = 13.30$), we see that the predictions follow the actual values quite well.

Figure 16: Comparison of different training trips from the Modified DeepONet.

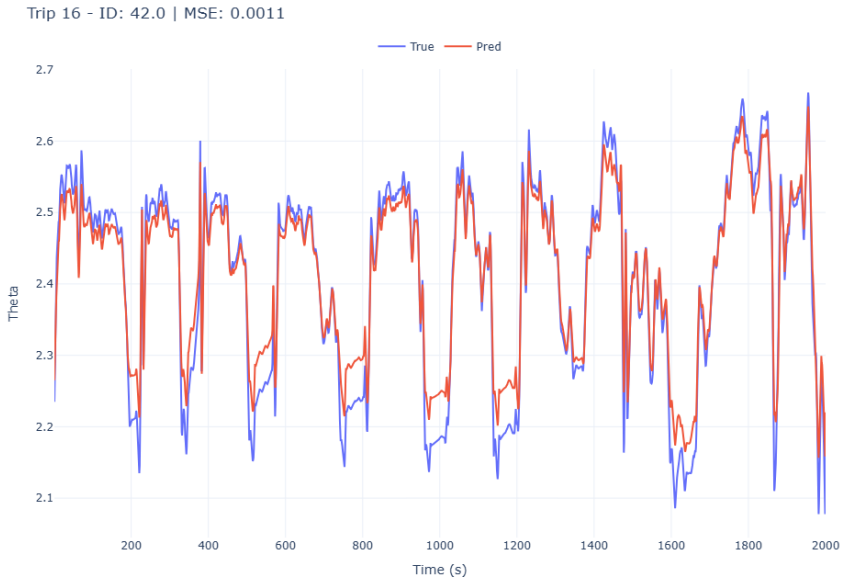
The behavior seen from the trip in fig. 16a is seen more often in other trips,

whereby it overshoots the predictions slightly. For all evaluation metrics (R^2, RMSE and MAE scores), the model outperformed the baseline on every trip.

6.1.2 Test set

For the test set, we again present results for both an easier and a more challenging trip. The model performs well overall, accurately capturing the general pattern of the signal. However, similar to the training data, a consistent offset is observed in the predictions. This offset can be interpreted as a local bias: a systematic, region-specific deviation between the predicted and actual signal that arises when the model's local encoding introduces context-dependent shifts in the output. This is expected, as local biases tend to be more significant on unseen data where the model has not previously encountered similar conditions. These local biases may stem from the way the architecture was encoded with the local trunk network, but could also be the results of sensor values drifting, or something occurring within the dredging system (for example interference in the dredging operation of debris).

(a) Predictions for the easier test trip ($V(\theta) = 2.45$). We see that the prediction overshoots slightly at most points, but that the overall pattern is captured well.



(b) Predictions for the test trip with more fluctuations ($V(\theta) = 18.78$), we see that the model predicts the pattern well, with some overpredictions in the troughs of the trip.

Figure 17: Comparison of different training trips from the Modified DeepONet.

In the more volatile trip shown in fig. 17b, the model captures the overall pattern well until it encounters sharp drops at the troughs. While it manages to track these decreases, it does so with a consistent offset and a tendency to overpredict. Across the entire trip, the model exhibits varying degrees of offset. This behavior is expected, given the fluctuating nature of the trip, which poses a greater challenge for accurate prediction. Additionally, both testing trips struggle to capture the pattern in the final 100 seconds of their respective trips, which is common for the other trips as well (fig. 18). Across the entire test set, the model outperforms the baseline in terms of RMSE and R^2 for all 17 trips. Averaged over all trips, the RMSE is reduced by 73.5% and the R^2 improves from 0.2647 to 0.9416. For the MAE, it achieves better performance than the baseline on 16 out of 17 trips, with an average improvement of 57.1%.

We show an RMSE heatmap for every timestep corresponding to every test trip. We see that many trips have higher errors at the start and end of a trip. This is unsurprising due to how we set up the training in the form of windows with a moving stride, so the start and the end timesteps of a trip would have the least amount of data to train on. With the window size being 100 and the stride as 10, timestep $t_1 \dots t_{10} = 1s \dots 10s$ is only prevalent in one training window. Timesteps $t_{11} \dots t_{20}$ only in two training windows, etc. The final 100 timesteps, $t_{1901} \dots t_{2000}$ are all only in one training window since we do not consider windows with a size smaller than 100.

The start and end of a trip also signifies the time when dredging is starting up (initializing) or shutting down, making it difficult to predict, and also, less important. Most trips have a relative constant RMSE throughout the trip, whilst other trips exhibit a highly fluctuating error (like in trip with ID 60.0).

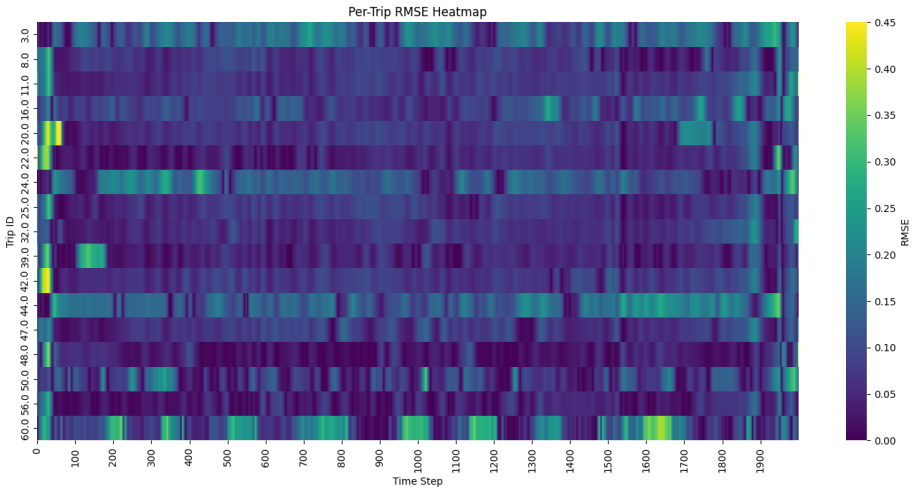


Figure 18: Heatmap of RMSE errors on test trips.

6.2 Performance with corrector model

We perform a search to inspect which correction window size $c_w \in [1, 50]$ offers the most improvement (lowest average RMSE per trip) on the test set. The search yielded $c_w = 1$ as seen in fig. 19, which we will now use for all future purposes. The powerful effect of the corrector model is that it corrects the offset that is caused by local biases from the model. This is reflected by the improvements in the RMSE. On the training data, averaged per trip, RMSE improved from 0.0124 to 0.0066. The RMSE on the test set improved from 0.0148 to 0.0094. From fig. 20 it is evident that the RMSE improved for every single trip. It does not improve by the same magnitude for every trip, especially for trips with high fluctuations. It does however, significantly improve the quality of predictions of trips suffering from a local bias offset.

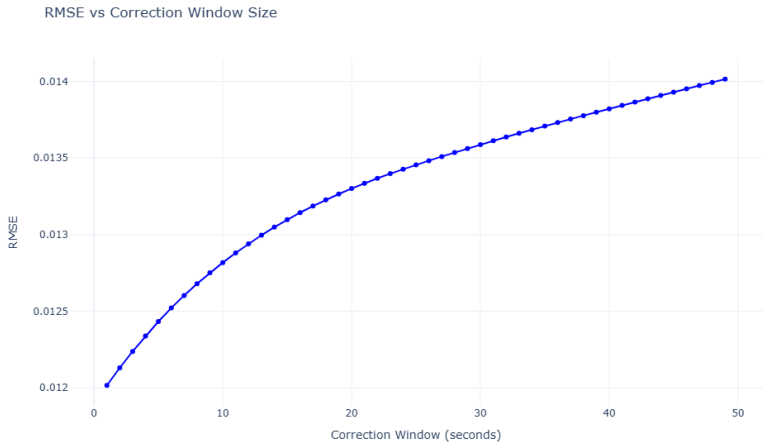
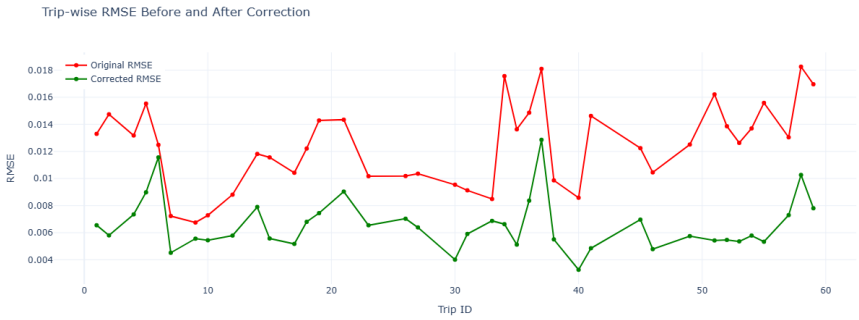
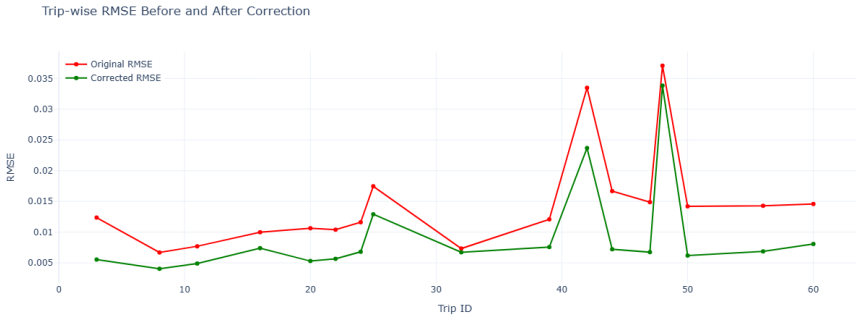


Figure 19: Effectivity of different correction windows on the average RMSE over every testing trip. We observe that longer correction windows result in higher RMSE values. This behaviour can be attributed to the temporal nature of the correction mechanism. Since the correction is applied based on past errors, including a larger temporal span introduces outdated information that no longer reflects the current system state. In contrast, shorter windows, particularly $c_w = 1$, focus on the most recent deviations. This is more representative of the present dynamics.



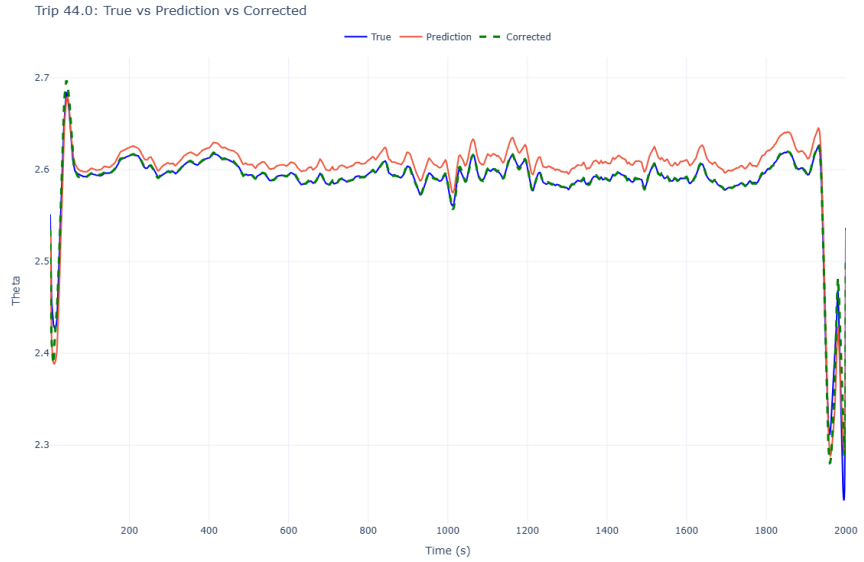
(a) Corrector model improvements on the training trips.



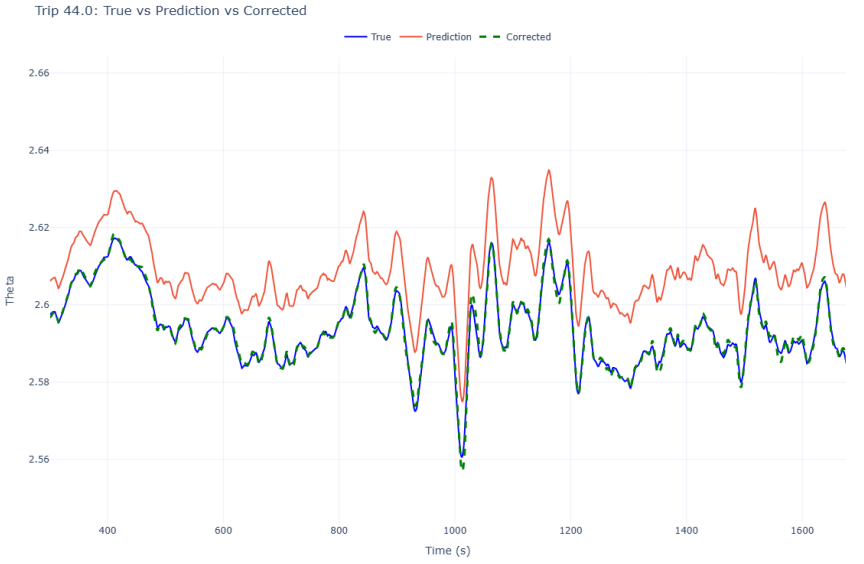
(b) Improvements by the corrector model on the testing trips.

Figure 20: RMSE improvements.

It is evident that all trips consistently improve after applying the correction. Particularly in the testing trips, the improvements seem to be by the same amount for every trip. There are no trips that improve drastically, i.e., the high error trips (such as trip with ID 46), remain with a high error also after correction. For the training trips, however, we see that the trip with ID 34 and 35 did improve by a considerable amount. This implies that this trip suffered from a persistent constant offset instead of large variability, which is adjustable by the correction implementation. We showcase the improvement from the bias offset of trip 2 (ID 44.0) in fig. 17a and observe how the corrected prediction now follows the actual values seemingly perfect in fig. 21.



(a) Corrected, true and original predictions for trip 2 with ID 44.0. Corrected predictions seem to follow the true values perfectly, correcting the bias from the original predictions. The end of the trip is still not predicted accurately, even with the addition of the corrector mechanism.



(b) The same trip but zoomed into timesteps 400 to 1600. We can clearly see how the predictions benefit from the correction.

Figure 21: Predicted, corrected, and true θ values for trip with ID 44.

Regarding comparisons to the baseline, the corrected model outperformed the baseline on every single metric, for every single trip (both on testing and training trips). The modified net (ModNet) also clearly outperforms the classical DeepONet, as well as the MIONet. We analyse the results of these models in appendix B. We summarize the results in table 5 and table 6 below.

Table 5: Performance metrics for the training set.

Architecture	**Training time (hours)**	**RMSE**	**MAE**	$\mathbf{R^2}$
ModNet	8	0.0124	0.0106	0.9589
Corrected ModNet	8	0.0066	0.0032	0.9886
Baseline	0	0.0577	0.0354	0.3082
MIONet	3	0.0460	0.0328	0.2587
Classical DeepONet	1	0.1039	0.0883	-1.4724

Table 6: Evaluation metrics across architectures on the testing set. Interestingly, the classic DeepONet performs better on testing trips than training trips (more details in appendix B).

Architecture	**RMSE**	**MAE**	$\mathbf{R^2}$
ModNet	0.0148	0.0115	0.9416
Corrected ModNet	0.0094	0.0036	0.9745
Baseline	0.0592	0.0359	0.2647
MIONet	0.0665	0.0487	-0.0449
Classical DeepONet	0.1003	0.0855	-1.4015

We plot a histogram of the RMSE performances across all testing trips in fig. 22. It is evident that both the corrected, as well as the uncorrected model instants outperform the baseline for all trips. In appendix C, the visualizations for every individual trip in the testing set is presented.

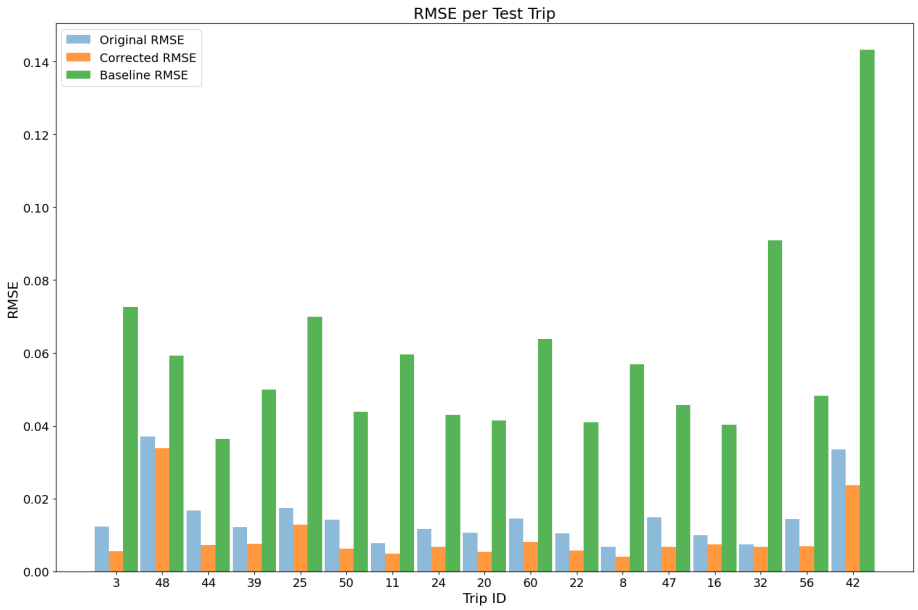


Figure 22: Histogram of RMSE scores for simulated testing trips. We also include the RMSE scores of the baseline model.

6.3 Sensitivity analysis

To examine the robustness of the model we investigate the temporal robustness: this refers to how well the model performs when evaluated on data from a different time period it was trained on. Feature importance is evaluated through dropout analysis, where we systematically remove one input at a time and measure the degradation in performance. For each feature we retrain the model with identical settings (architecture, optimizer, hyperparameters, and splits) but with that feature excluded, and we record the evaluation metrics for that model. This provides a transparent ranking of inputs.

6.3.1 Temporal robustness

When investigating temporal robustness, we investigate model's ability to generalize across different periods in time. This is particularly crucial in time-dependent processes such as dredging. There are external factors including

weather conditions, sea states, sediment properties, and vessel loading patterns that can significantly influence system dynamics. So far, the model has only been trained and tested on 56 simulated trips that are derived from sensor data within the same month. This could mean that the model is only accurate for that specific dredging context for that month. Hence why we should test to see if it can still perform under a different context, at a different time.

To evaluate temporal robustness, we use our trained model that was optimized for simulated trips in November 2023, and test it on simulated trips that occured in June 2023 (43 trips), as well as January 2024 (55 trips). The construction of simulated trips is done with the same process as is shown in section 5. Performance is measured on trips using the standard metrics.

A temporally robust model should be consistently accurate across different operational periods, even when the system experiences shifts due to seasonal variations or changes in soil types.

We showcase the performance across all metrics in table 7.

Table 7: Evaluation metrics across different temporal datasets. Model was trained on data from November 2023.

Dataset	**RMSE**	**MAE**	$\mathbf{R^2}$
November 2023 (unseen testing trips)	0.0148	0.0115	0.9416
January 2024	0.0350	0.0116	0.9096
June 2023	0.0207	0.0117	0.8881

Surprisingly, we see that the dataset from January 2024 and June 2023 both perform extremely well. The January dataset performs marginally better than the June dataset in terms of MAE and R^2 scores. The area in which the vessel Strandway was dredging was similar in January as well as June. This area was also part of the training trips in November. This explains why the model performs so well for both timespans (as seen in figs. 23 and 24).

Figure 23: Prediction and actuals from a simulated dredging trip in January 2024 ($V(\theta) = 5.99$). The predictions seems to follow the actuals almost exactly. Only in the last quarter of the trip there is a visible bias offset, with the performance degrading over time.

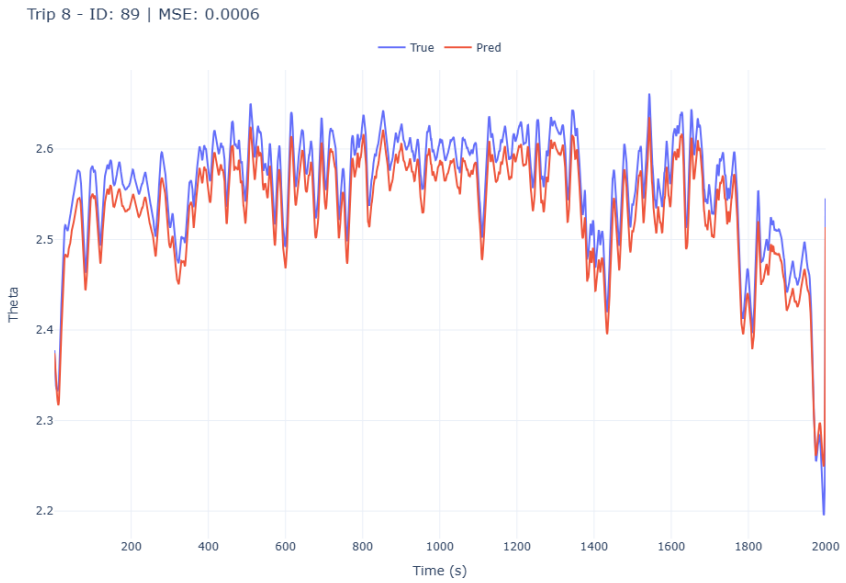


Figure 24: Predictions of a dredging trip that took place in June 2023 ($V(\theta) = 9.45$). The predictions follow the actual values closely. The behaviour of θ in this trip is notably erratic and irregular. Usually, the behaviour across one trip is relatively smooth. This indicates that during this time period, a different sediment such as water could have been dredged, instead of sand (which was trained on). Even though the behaviour is erratic, the model still generalizes well.

For this analysis, we intentionally disabled the correction module to assess the robustness of the modified DeepONet. Across both figures, we again observe systematic offsets in segments of the predictions, indicative of a persistent bias. These offsets can be mitigated by enabling the correction methodology, which is designed to compensate for slowly varying bias using recent error feedback. We therefore expect an improvement in predictions once corrections are applied.

6.3.2 Dropout analysis

To evaluate the relative importance of each input feature to the operator learning model's predictive performance, we conduct dropout analysis. This approach systematically removes one input feature at a time from the model's

branch inputs, retrains a new model, and evaluates its performance on unseen test data. The degradation in prediction metrics (RMSE, MAE, R^2) provides an estimate of that feature's contribution to model accuracy.

Let $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$ denote the set of input functions (branches). Each model uses a local trunk formulation with window size 100 and stride length ten. This data is structured exactly the same as the training data used for the original model. The hyperparameters and interactions between branch and trunk networks used to train the individual models in the dropout analysis remain consistent with the ones used to train the main model.

For each branch feature $f_i \in \mathcal{F}$, we conduct the following steps:

1. **Drop feature:** Exclude f_i from the branch input set, forming a reduced set $\mathcal{F}_{-i}$ with $d = |\mathcal{F}_{-i}|$.

2. **Model training:** A modified DeepONet model with d branch subnetworks and a local trunk network is instantiated. The model is trained using mean squared error (MSE) loss for ten epochs.

3. **Prediction and evaluation:** The trained model is evaluated on the test set. The corrector model is then also applied to inspect the performance of the corrected predictions on the test set. The usual RMSE, MAE, and R^2 metrics are recorded. The average of the evaluation metrics across all test trips is saved. This is done once for solely the model, and once with the corrector model applied.

This analysis provides insight into the influence of each feature on the model's prediction quality and highlights redundancies or critical dependencies in the input data.

We showcase the results of the analysis in tables 8 and 9 below.

Table 8: Dropout analysis: evaluation metrics (uncorrected) after removing individual features. Higher RMSE/MAE and lower R^2 indicate greater importance of the dropped feature.

Dropped feature	**RMSE**	**MAE**	$\mathbf{R^2}$
Jet pumps	0.0647	0.0574	-0.1814
ΔP_{Head}	0.0990	0.0507	-1.9645
Vacuum pressure	0.0559	0.0452	0.1741
Mixture velocity	0.1319	0.1258	-3.5512
Draghead depth	0.0286	0.0140	0.7916
Angle	0.0234	0.0139	0.8571
Swellcompensator	0.0499	0.0317	0.3379
Lagged density	0.0761	0.0713	-0.4345
All features	0.0124	0.0106	0.9589

Table 9: Dropout analysis: evaluation metrics (corrected) after removing individual features. Correction applied via rolling mean alignment.

Dropped feature	$\mathbf{RMSE_{corr}}$	$\mathbf{MAE_{corr}}$	$\mathbf{R^2_{corr}}$
Jet pumps	0.0317	0.0115	0.7285
ΔP_{Head}	0.0796	0.0279	-0.9629
Vacuum pressure	0.0391	0.0184	0.6039
Mixture velocity	0.0376	0.0197	0.6757
Draghead depth	0.0272	0.0090	0.8067
Angle	0.0215	0.0070	0.8653
Swellcompensator	0.0392	0.0145	0.5006
Lagged density	0.0284	0.0095	0.7719
All features	0.0066	0.0032	0.9886

From tables 8 and 9, the largest degradation in performance occurs when ΔP_{Head}, mixture velocity, or lagged density are removed. By contrast, dropping angle or draghead depth has the smallest effect, and no model matches the

performance of reference model that uses all features.

These results are consistent with the relations discussed in section 2.2. In that section we argued that the target θ depends on the entrance loss facor α, which in turn is driven by the suction conditions at the draghead. The pressure change across the draghead, ΔP_{Head}, is an indication for that suction: changes in ΔP_{Head} modulate α and therefore the mixture composition. The mixture velocity and mixture density constitute two of the most important parameters in the dredging process as a whole, which is reflected in the results.

In contrast, angle and draghead depth exert a weaker influence: their effects on θ are largely mediated through the other features such as ΔP_{Head} and vacuum pressure, and their variation is limited. Taken together, the fact that every feature drop degrades performance indicates that each variable contributes complementary information needed for an accurate predictor, with the best performing model being the one with all features.

Seeing as we simulated the data from eq. (6) in section 5, these results make sense. Whilst simulating the data, we delegated specific weights for each variable signifying their importances. This is reflected back in the dropout analysis, where the variables given the most weight was $\Delta \text{P}_{\text{Head}}$, and the one given the least weight were both the jet pumps and the angle. This analysis further proves that the model captures the patterns of the simulated system. In the next (sub)section, we explore the performance of our model on real data, where the importances are unknown. Here we repeat the dropout analysis.

6.4 Evaluation on real data

We conduct experiments using real-world data. Instead of training on simulated values of θ, we now compute and use real values derived from sensor readings. The input variables are processed identically to the simulated setup. First, a dynamic rolling window is computed to temporally align the density sensor with the other sensors. Based on this aligned dataset, the θ values are calculated using the vacuum equation. To smooth the output, a rolling mean

with a window size of 100 is applied to the computed θ, followed by a Savitzky-Golay filter [49] with a 50-step window and polynomial order of three, applied to both input and output variables.

Outlier removal is performed by clipping all θ values outside the range $\theta \in [0, 5]$. This means that any values outside the range are clipped to 0 or 5. Any trips for which more than 10% of the data is clipped are excluded from analysis. Out of 56 available trips from the November dataset, 32 met the criteria for evaluation. This is about 60% of the trips, which demonstrates the instability of using real data. The dataset is split in the same manner as before: 70% of the trips are used for training and 30% for testing (randomly chosen). Trips are segmented into windows of 100 seconds with a buffer (stride) of ten seconds. The architecture of the branch and trunk networks remain unchanged, with the only modification being an increase in the number of training epochs from 100 to 200 due to slower observed convergence and less training data. Training the model on this dataset required approximately 9 hours. After training, we apply the corrector model with a correction window of $c_w = 1$.

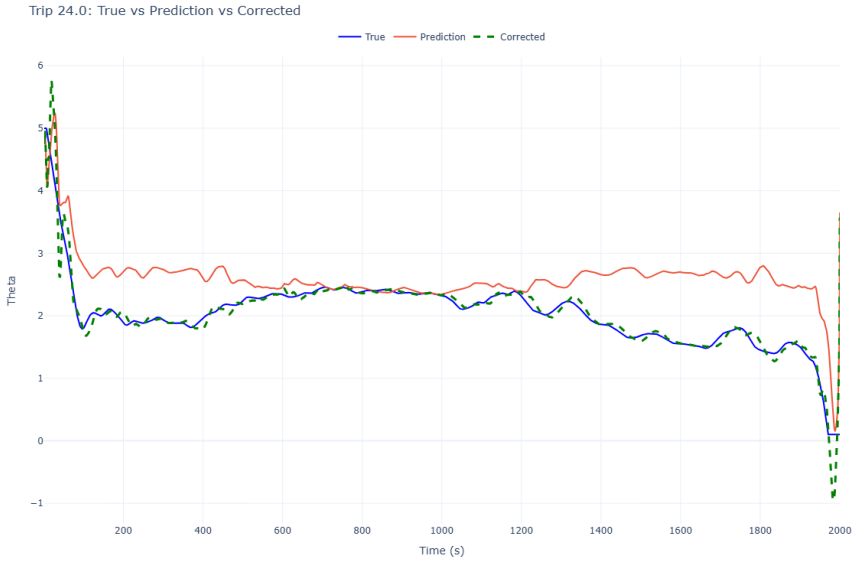


Figure 25: Corrector model correcting local biases on a testing trip of real data ($V(\theta) = 10.03$). The modified DeepONet prediction suffer from varying amounts of local biases, which is solved with the addition of the corrector model.

The results indicate that the model performs less effectively on real data compared to the simulated case. This outcome is expected, given the higher noise levels and unpredictability inherent in real-world sensor readings. While the initial model's performance on the test set is limited, the introduction of the corrector model significantly improves the results by addressing local prediction biases. Although local bias was only a minor issue in the simulated setting, it becomes a significant challenge when working with real data.

It is important to note that the model and training procedure were extensively tuned for the simulated dataset. With further (hyperparameter) tuning specific to real-world conditions, the model's performance could improve. Furthermore, only 60% of the trips were valid, consequently causing there to be less data. Perhaps with more valid trips to train on, performance may also improve. An example of a test set prediction, with and without correction, is shown in fig. 25. The figure illustrates the presence of local bias in the raw predictions

and the corrective effect introduced by the corrector model.

Not all predictions suffer from local biases, with some outputting high errors in general. Overall on the testing set, the modified DeepONet with the correcting mechanism outperformed the baseline on 9/10 trips on the RMSE, R^2 and MAE metrics. For the training data, the corrected model outperformed the baseline on all 22 trips for the RMSE, R^2 and MAE metrics. We summarize the metrics for the real data in tables 10 and 11 below.

Table 10: Evaluation metrics for real training data. For the MIONet and the classical DeepONet, the average prediction per time step was used for evaluation, since this was more accurate than the first prediction.

Architecture	**RMSE**	**MAE**	$\mathbf{R^2}$
ModNet	0.7312	0.6428	-1.3518
Corrected ModNet	0.2241	0.1284	0.8159
Baseline	0.3932	0.2504	0.4583
MIONet (avg)	0.8371	0.7031	-1.9146
Classical DeepONet	0.8650	0.7367	-2.0650

Table 11: Evaluation metrics for real testing data. We keep in mind that there was one trip in the testing set that performed extremely poorly, hence why the large average error is present in the R^2 value of the uncorrected ModNet.

Architecture	**RMSE**	**MAE**	$\mathbf{R^2}$
ModNet	0.9988	0.8517	-2.6209
Corrected ModNet	0.2917	0.1842	0.7182
Baseline	0.3880	0.2497	0.5394
MIONet (avg)	0.9189	0.7772	-2.0244
Classical DeepONet	0.8906	0.7401	-1.8179

We illustrate the performance on the testing trips with the histogram in fig. 26. We can see how one trip (ID 35) perform considerably worse than the others. This is also the only trip where the baseline outperforms our corrected

model. This trip is illustrated in fig. 27. The trips with ID 44, 58, 49 and 24 can be seen to have the largest improvement from the corrector model, showcasing the effectivity of the corrector model at dealing with local biases.

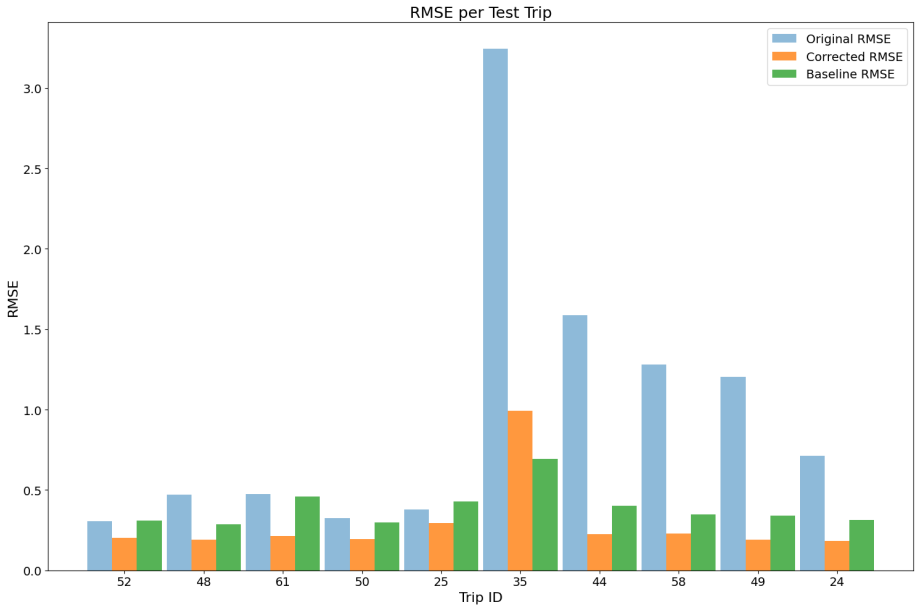


Figure 26: Histogram of RMSE scores for real testing data.

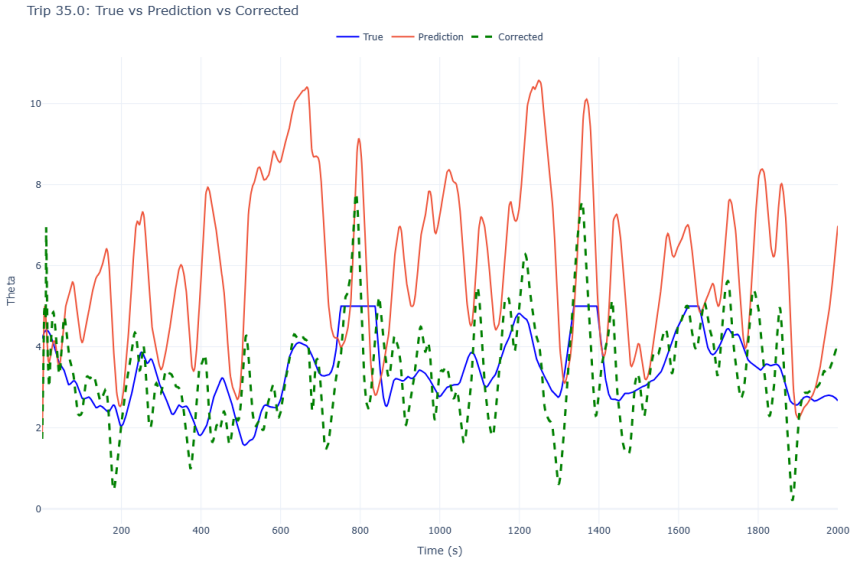


Figure 27: Predicted and real θ values for the only trip where the baseline outperforms the model ($V(\theta) = 37.91$). The flat areas in the true values is when θ was clipped to $\theta = 5$ whenever $\theta > 5$. We see that both the initial prediction, as well as the corrected one struggle to capture the pattern. This shows that the base predictions are important, the corrector model can not be used on its own.

In fig. 27, a trip that may have suffered from external factors, due to the large values of θ is presented. According to deliberation with an expert (R. Higler, personal communication, April 2025), such large values indicate, that in this trip, the draghead may have been clogged. The draghead may have dredged large rocks, which block the draghead, causing large pressure differences, which could explain the high values of θ. This type of behaviour was not present in any of the trips that the model was trained on, so it comes as no surprise that it struggles to capture the intricacies of a clogged head. This highlights the importance of choosing the right trips to train on. In practice this can be done by individually analysing and collecting trips which we are confident possess non-anomalous behaviour. Again, we see that training and testing with real data is challenging and unpredictable, as opposed to a simulated system.

6.4.1 Dropout analysis on real data

We repeat the exact same process as in section 6.3.2, but this time for the smoothed real dataset. The data was not simulated and controlled beforehand, so the results indicate what the model believes are the most important features. The results of the dropout analysis are visible in tables 12 and 13.

Table 12: Dropout analysis: evaluation metrics (uncorrected) after removing individual features. Higher RMSE/MAE and lower R^2 indicate greater importance of the dropped feature.

Dropped feature	**RMSE**	**MAE**	**R^2**
Jet pumps	0.7306	0.6253	-1.0171
ΔP_{Head}	0.9551	0.8088	-2.1533
Vacuum pressure	0.7623	0.6427	-1.1969
Mixture velocity	0.7636	0.6523	-1.1661
Draghead depth	0.8261	0.7359	-1.5616
Angle	0.8207	0.7225	-1.5479
Swellcompensator	0.8715	0.7978	-1.8928
Lagged density	0.7286	0.6281	-1.0188
All features	0.9988	0.8517	-2.6209

Table 13: Dropout analysis: evaluation metrics (corrected).

Dropped feature	$\mathbf{RMSE_{corr}}$	$\mathbf{MAE_{corr}}$	$\mathbf{R^2_{corr}}$
Jet pumps	0.2356	0.1461	0.8300
ΔP_{Head}	0.3226	0.1478	0.6679
Vacuum pressure	0.2246	0.1406	0.8427
Mixture velocity	0.2126	0.1238	0.8576
Draghead depth	0.2582	0.1502	0.7955
Angle	0.2394	0.1475	0.8250
Swellcompensator	0.2190	0.1421	0.8530
Lagged density	0.2237	0.1387	0.8459
All features	0.2917	0.1842	0.7182

Also when using real data, it is evident that the most important feature is the ΔP_{Head}, since model performance diminishes the most when removing that feature. This aligns with the theory, as well as with the analysis from the simulated data. Besides the ΔP_{Head}, there is not another variable that stands out. For the uncorrected predictions the results point towards the swellcompensator, whilst the corrected predictions indicate the importance of the draghead depth. Interestingly enough, the draghead depth was concluded to be one of the least important variables according to the dropout analysis on the simulated data. This shows that there are still some inaccuracies in the way that the data was simulated, which is expected. The dropout analysis results show us that these inaccuracies have to do with misjudging the importance and relevance of a number of variables such as the mixture velocity and vacuum pressure. In future work, this dropout analysis can be used as a basis to rework the simulated system to better reflect dredging dynamics.

An interesting observation is that all the evaluation metrics taken from the dropout analysis (apart from the one where ΔP_{Head} is taken out) are better than the main model performance. This indicates that the main model struggled with the training data. The main model was trained for 200 epochs, whilst the

ones in the dropout analysis were trained for just ten epochs each. Removing an additional input variable also reduces the complexity of the model with less parameters to optimize. This indicates a superior generalization ability. When examining the individual trips we see that this is indeed the issue. We inspect the trip that performed poorly in fig. 27, but for the model that trained without the swellcompensator variable. The (uncorrected) prediction is seen in fig. 28. It is evident that the predictions for the trip are much closer to the actuals, and the majority of the pattern is captured. It does not exhibit any rapidly changing (eccentric) behaviour such as in fig. 27. This demonstrates the improved generalization of this specific model.

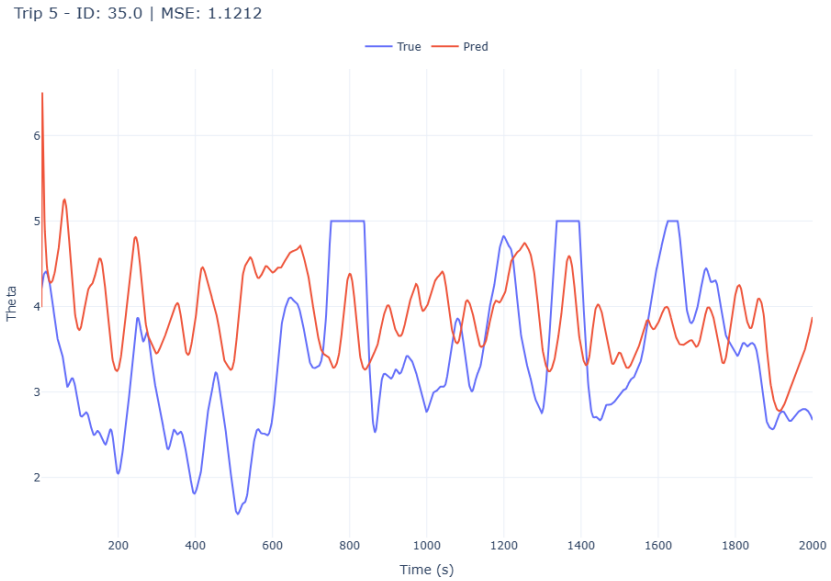


Figure 28: Prediction and actuals for trip ID 35 on model removing swellcompensator as input. It generalizes better than the main model in fig. 27.

6.5 Suction production simulation

The aim of the project was to optimize the production of a TSHD. We run simulations to inspect how much production we could have saved if the model

would have been used. Due to data sensitivity reasons, we present this section using the simulated dataset.

With the available sensor dataset, we calculate the production that would have happened with the simulated value for θ. From the simulated θ values and the corresponding sensor data, we calculate the simulated production from the production formula in section 2.3:

Calculating the predicted mixture density in the pipe using the simulated θ:

$$\rho_m(V, d_1, v, \theta_{sim}, d_2) = \frac{V + d_1 \rho_w g}{\frac{\theta_{sim} v^2}{2} + (d_1 - d_2) g}$$

Plugging this into our production equation we end up with:

$$Prod_{sim}\left(m^3/s\right) = \left(\rho_m(V, d_1, v, \theta_{sim}, d_2) - \rho_w\right) \frac{v}{\rho_s - \rho_w} \frac{\pi}{4} D^2 \tag{7}$$

The optimal velocity (i.e., optimal point on the curve) for which the vessel should dredge to output the maximum production (for a constant vacuum) is then calculated as:

$$v' \text{ such that } \frac{\partial Prod_{sim}}{\partial v}(v') = 0$$

This then gives us the maximal achievable production

$$\overline{Prod_{sim}} = Prod_{sim}(v')$$

To attain values for the production, we again work with windows. Since every ten seconds, we get new predictions for the past ten values of θ, we calculate production in windows of ten seconds, where the values are averaged over the ten second window. This way, every ten seconds, the crew can see where they are on the vacuum curve.

To find out the amount of potential production that was missed out with not using the model, we generate the production curve using the predicted values of θ, for $v \in [1.01, 1.02, \ldots 20]$. The mixture velocity whilst dredging never exceeds these bounds (R. Higler, personal communication April 2025).

$$Prod_{pred}\left(m^3/s\right) = \left(\rho_m(V, d_1, v, \theta_{pred}, d_2) - \rho_w\right) \frac{v}{\rho_s - \rho_w} \frac{\pi}{4} D^2$$

The predicted optimum is then calculated by simulating all points in the range $v = [1.01, 1.02, \ldots 20]$ such that we have the maximum achievable (predicted) production from velocity $\tilde{v}$:

$$\overline{Prod_{pred}} = Prod_{pred}(\tilde{v})$$

We give an illustrative example of how this would look like in real-time in fig. 29.

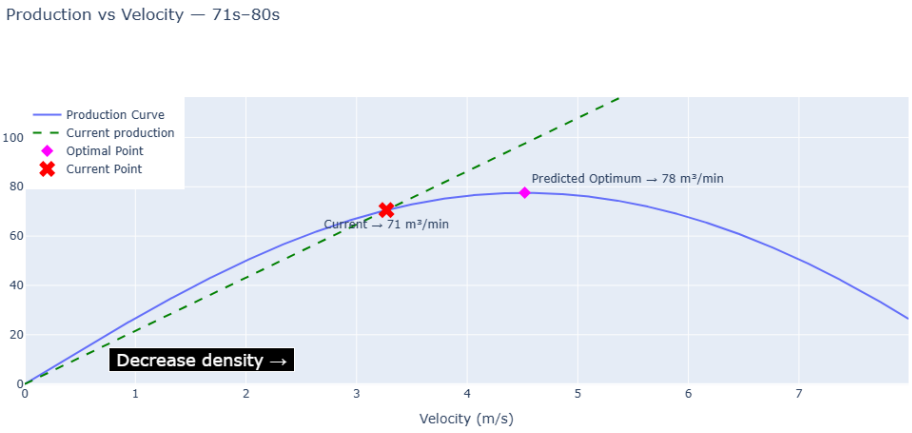


Figure 29: Production simulation of trip with ID 42, from $t_{71} \rightarrow t_{80}$. The red cross indicates where the vessel is dredging currently, which is calculated from the predicted value of θ. This comes from $Prod_{pred}$ in eq. (7), averaged over the window of ten seconds. The pink point indicates the calculated optimal production from the predicted θ : $\overline{Prod_{pred}}$. The current production being on the left side of the optimum indicates that the density should be decreased, causing a flatter slope of the density line in green. This will push the production closer to the optimum. Decreasing the density can be done by adding more water to the mixture through the relief valve.

We now calculate the potential production that was missed out with the following methodology:

For every testing trip, for every window of ten seconds, we calculate the

difference:

$$min\left(\overline{Prod_{pred}}, \overline{Prod_{sim}}\right) - Prod_{sim}$$

This is because the predicted optimal point can never exceed the theoretical optimum. This is done for every window in every testing trip, where the results are then aggregated. The results showed that the dredging vessel missed out on $1,019.24m^3$ of production over 17 trips, which would have been an average of an 8.52% increase in production per trip. We also checked the accuracy of the advice, that is, whether the crew should increase or decrease the density. We compare the advice given when using the predicted θ, with the advice given when simulating production using the simulated θ, over every time window in the 17 trips. Using the predicted values for θ proved to be accurate 99.88% of the time.

These calculations are made using the perfect conditions. It is assumed that the crew is able to get to the optimal point in that ten second time window and then stays on it. Whilst the vacuum relief valve can be opened to decrease the density of the mixture, increasing the density is more complex. This can be done by moving the draghead but it is not as easy to control the exact amount. Our assumption is that crews aim to operate near the production optimum. In practice, operational objectives can be broader: crews may target sub-optimal points for logistical or strategic reasons (e.g., schedule constraints, coordination with other vessels, or competitive dynamics) (W. de Graaf personal communication, April 2025). The modelling framework remains useful in these settings because it characterizes the full production curve rather than a single optimum. Consequently, fleet managers can select any desired operating point, e.g., a target corresponding to 75% of the optimum. This flexibility is a strength of the approach: it supports decision-making under multiple objectives, not only maximum production.

7 Conclusion

We presented a data-driven and physics-guided framework based on operator learning to estimate the loss parameter θ within the vacuum process of Trailing Suction Hopper Dredgers. The accurate prediction of this parameter has significant value, as it facilitates real-time suction production optimization and enhances the reliability of sensor systems.

We began by identifying the key physical processes governing the vacuum operation and derived relations for the input variables with θ. A synthetic dataset was constructed using real-world sensor readings and domain-informed equations, enabling controlled experimentation and model validation. This domain insight informed model structure, input selection, and training procedures.

Building on the DeepONet architecture, we introduced a modified operator learning model incorporating attention-based interactions between branch and trunk networks. A local trunk mechanism was introduced to improve modeling of localized temporal dynamics: a necessary adaptation due to the variability between individual dredging trips. Our architecture outperformed classical DeepONet and baseline methods across multiple evaluation metrics, including RMSE, MAE, and R^2.

To mitigate persistent local biases, a corrector model was designed using a rolling mean error correction strategy, further improving performance. This model uses lagged sensor signals to dynamically adjust predictions. This hybrid approach: combining simulation-informed predictions with real-time correction mechanisms, improved model robustness, and enhances generalization to real-world conditions. The operator learning architecture accurately predicts for simulated data, as well as for real data.

The corrector model also enables real-time anomaly detection by continuously analyzing the residuals between predicted and observed sensor values. By incorporating lagged sensor signals and tracking deviations, the model can identify inconsistencies that indicate potential sensor failures. This supports proactive monitoring of sensor health, allowing for timely interventions. As a

result, the overall reliability of the dredging process is enhanced, reducing the risk of production inefficiencies or decision errors. Therefore, the integration of this real-time mechanism not only strengthens model accuracy, but also adds a maintenance benefit.

Furthermore, we demonstrated the robustness of the proposed framework through temporal testing and dropout analysis. Under different timespans, the model continued to perform exceptionally well. Dropout analysis further confirmed that the model was guided by the underlying physics. According to theoretical understanding, the primary driver of θ is ΔP_{Head}, which emerged as the most influential feature in both the real and simulated models. Beyond this, the model showed limited reliance on other features (for real data), showing its robustness with respect to input selection.

To address the research questions directly: the operator-learning framework predicts the loss parameter θ with high accuracy on simulated data, and this accuracy largely transfers to real-world streams once the corrector model is implemented. Embedding physics into the architecture via domain-guided inputs, attention-mediated interactions, and a local trunk improves both predictive performance and interpretability relative to the classical DeepONet and non-operator baselines; in particular with dropout analyses aligning with the theory.

In deployment, the base operator model trained on simulation exhibits modest local biases; the rolling, lag-based correction closes this gap without retraining the operator, demonstrating improvements across all evaluation metrics. Leveraging residuals enables real-time sensor-health monitoring, where threshold-based criteria detect anomalies early enough to enable corrective action. Finally, the hybrid design: training on simulation while adapting online through lagged-sensor corrections proves more robust than purely simulated or purely data-driven approaches. Taken together, the experiments show that the architecture exhibits robust suction-production estimates. The operator provides a stable production curve. In evaluations over 17 test trips, the guidance derived from the predicted θ would have recovered $1,019.24m^3$ of production in

total, an average 8.52 % gain per trip. These outcomes indicate that the hybrid approach preserves decision quality by providing estimates that are reliable for continuous selection along the production curve, even when conditions deviate from the ideal setting.

In conclusion, this thesis presents a practical approach to integrating physics-based insights into operator learning for dredging systems. This practical approach offers the potential for increases in production, as well as a reduction of emissions.

8 Future Research

While the present work demonstrates that operator learning can be successfully applied to predict the loss parameter θ in the vacuum process of Trailing Suction Hopper Dredgers, several directions remain open for further study.

Model and Training Extensions

A first avenue concerns the refinement of the model architecture. The attention mechanism introduced here can be extended with more expressive variants such as multi–head or transformer-based attention layers [58]. These could allow the model to learn richer interdependencies between input variables. On the first of September of 2025, one month before the completion of this thesis, a paper was published about this topic [62]. The paper discusses different methodologies for combining the transformer self-attention mechanism with DeepONets. This research can be compared and applied to this thesis in future works. Another extension is to include adaptive learning rate schedules [70] or optimizers that automatically balance the back–propagated gradients of the multiple branch networks.

Regularization also offers potential improvement. Incorporating normalization layers, L_1 or L_2 weighted penalties, and dropout [17] could enhance generalization when training on noisy sensor data. These additions would help

distinguish between physically meaningful fluctuations and measurement noise, an aspect that becomes crucial when transferring the model to real data.

Regarding the modified DeepONet architecture presented in this thesis, several further experiments can be conducted. The preprocessing procedures may be adjusted to explore both stronger and weaker smoothing using the Savitzky–Golay filter (or even removing filter altogether). In the current dataset the lagged density was configured to be a constant of ten seconds. In the future, the architecture can be tested on density that lags by varying amounts to test robustness. Whilst a simple corrector model was implemented as a rolling error correction, a more sophisticated methodology can be developed to incorporate lagged values into the architecture itself. The lagged true values could, for instance, alter parameters of the network based on the behaviour of a trip.

The interaction dictionary could be reconfigured to examine how the model behaves when all input variables interact, rather than only a selected subset. Another opportunity for testing comes with adapting the high rank version of the MIONet with interactions, rather than the low rank version. Finally, the hyperparameter optimization process could be repeated and re-tuned specifically for the real data, ensuring that the configuration is better aligned with the noise characteristics and variability present in practice. More hyperparameter configurations such as different layer sizes for the different branch and trunk subnetworks can also be tested in this process.

Data and Robustness Studies

A key limitation of the current study is the absence of controlled noise in the synthetic dataset. Introducing different levels and types of noise would make it possible to quantify how much of the model's performance stems from learning true physical relationships rather than memorizing smooth trends. Such tests could clarify whether the discrepancies observed on real data arise from sensor noise or from unmodelled physical phenomena. Similarly, expanding the training set to include data from different vessels, dredging sites, and operational

conditions would improve the model's generalizability.

Future experiments should also examine how the choice of preprocessing parameters-such as window size, stride, and interpolation length affects model robustness. Systematically varying these parameters could reveal trade–offs between temporal resolution and noise suppression. Combining real and simulated data through hybrid or transfer–learning strategies represents another promising path: simulated data provide physical consistency, whereas real data introduce variability and noise characteristics that drive generalization.

Physics–Guided Adaptation

In practice, the end goal is to predict the mixture density directly and to use it for computing the vacuum dynamics in real-time. Predicting the mixture density was not the main focus in this thesis, but remains an important step for suction production simulation. Moreover, future work could investigate freezing the parameters of the pre–trained DeepONet, as suggested in [66]. The frozen network could then serve as a fixed physical prior, onto which a corrector model adapts to new vessels or environmental conditions without requiring full retraining.

Practical Limitations and Validation

Several aspects remain challenging. There exists no supervised system for θ, and the physical relations used for simulation may not perfectly describe the dredging operations. Consequently, full validation of model accuracy will only be possible once reliable estimates of θ become available. Until then, noise injection on synthetic data provides a controlled way of testing model reliability.

Finally, careful selection of representative dredging trips for both training and validation will be essential. This ensures that the model learns from physically meaningful patterns rather than vessel–specific behaviour. By progressively extending the dataset, incorporating noise models, and refining the learning architecture, the proposed framework could evolve into a robust, real–time

predictive system applicable across fleets and operating conditions.

References

[1] S. Agarwal, S. Sharma, R. Suresh, M. H. Rahman, S. Vranckx, B. Maiheu, L. Blyth, S. Janssen, P. Gargava, V. K. Shukla, and S. Batra. Air quality forecasting using artificial neural networks with real time dynamic error correction in highly polluted regions. *Science of the Total Environment*, 735:139454, 2020.

[2] S. Bai, M. Li, R. Kong, S. Han, H. Li, and Q. Liang. Data mining approach to construction productivity prediction for cutter suction dredgers. *Automation in Construction*, 105:102833, 2019.

[3] S. Bai, M. Li, L. Song, Q. Ren, L. Qin, and J. Fu. Productivity analysis of trailing suction hopper dredgers using stacking strategy. *Automation in Construction*, 122:103470, 2021.

[4] Royal Boskalis. Boskalis academy. https://boskalis.plusport.com/scripts/login.aspx. Accessed: 20-11-2024.

[5] Royal Boskalis. Sustainability report 2023. 2023.

[6] J. H. Cavalcanti, T. Kovács, and A. Kő. Production system efficiency optimization using sensor data, machine learning-based simulation and genetic algorithms. *Procedia CIRP*, 107:528–533, 2022.

[7] T. Chen and H. Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, 6(4):911–917, 1995.

[8] X. Chen, B. T. Cao, Y. Yuan, and G. Meschke. Transfer learning based physics-informed neural networks for solving inverse problems in engineering structures under different loading scenarios. *Computer Methods in Applied Mechanics and Engineering*, 405:115852, 2023.

[9] Z. Chen, J. Ye, D. Wang, and Y. Hong-li. The numerical prediction of draghead motion of trailing suction hopper dredger in time domain. *Ocean Engineering*, 91:146–151, 2014.

[10] J. Chou and N. Ngo. Time series analytics using sliding window metaheuristic optimization-based machine learning system for identifying building energy consumption patterns. *Applied Energy*, 177:751–770, 2016.

[11] A. Daw, A. Karpatne, W. D. Watkins, J. S. Read, and V. Kumar. Physics-guided neural networks (pgnn): An application in lake temperature modeling. In *Knowledge guided machine learning*, pages 353–372. Chapman and Hall/CRC, 2022.

[12] S. Desai, M. Mattheakis, H. Joy, P. Protopapas, and S. Roberts. One-shot transfer learning of physics-informed neural networks. *arXiv preprint arXiv:2110.11286*, 2021.

[13] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri. Activation functions in deep learning: a comprehensive survey and benchmark. *Neurocomputing*, 503:92–108, 2022.

[14] M. Elad, B. Matalon, and M. Zibulevsky. Coordinate and subspace optimization methods for linear least squares with non-quadratic regularization. *Applied and Computational Harmonic Analysis*, 23:346–367, 2007.

[15] R. X. Gao and R. Yan. Wavelets. 2011.

[16] L. Gonon, A. Jentzen, B. Kuckuck, S. Liang, A. Riekert, and P. von Wurstemberger. An overview on machine learning methods for partial differential equations: from physics informed neural networks to deep operator learning. *arXiv preprint arXiv:2408.13222*, 2024.

[17] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[18] V. Grimm, A. Heinlein, A. Klawonn, M. Lanser, and J. Weber. Estimating the time-dependent contact rate of sir and seir models in mathematical epidemiology using physics-informed neural networks. *Electron. Trans. Numer. Anal.*, 56:1–27, 2022.

[19] B. Grimstad, V. Gunnerud, A. T. Sandnes, S. S. Shamlou, I. S. Skrondal, V. Uglane, S. Ursin-Holm, and B. Foss. A simple data-driven approach to production estimation and optimization. *SPE Intelligent Energy International Conference and Exhibition*, 2016.

[20] X. Han, J. Jiang, A. Xu, A. Bari, C. Pei, and Y. Sun. Sensor drift detection based on discrete wavelet transform and grey models. *IEEE Access*, 8:204389–204399, 2020.

[21] P. Jin, S. Meng, and L. Lu. Mionet: Learning multiple-input operators via tensor product. *SIAM Journal on Scientific Computing*, 44(6):A3490–A3514, 2022.

[22] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

[23] D. Kim, Y. Cho, D. Kim, C. Park, and J. Choo. Residual correction in real-time traffic forecasting. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 962–971, 2022.

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

[25] A. N. Kolmogorov and S. V. Fomin. *Introductory real analysis*. Courier Corporation, 1975.

[26] P. Kuffel, K. Kent, and G. Irwin. The implementation and effectiveness of linear interpolation within digital simulation. *International Journal of Electrical Power Energy Systems*, 19(4):221–227, 1997. Power Systems Transients.

[27] L. Kulanuwat, C. Chantrapornchai, M. Maleewong, P. Wongchaisuwat, S. Wimala, K. Sarinnapakorn, and S. Boonya-aroonnet. Anomaly detection using a sliding window technique and data imputation with machine learning for hydrological time series. *Water*, 13:1862, 2021.

[28] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems*, 2, 1989.

[29] M. Li, Q. Lu, S. Bai, M. Zhang, H. Tian, and L. Qin. Digital twin-driven virtual sensor approach for safe construction operations of trailing suction hopper dredger. *Automation in Construction*, 132:103961, 2021.

[30] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

[31] Z. Li, N. Kovachki, C. Choy, B. Li, J. Kossaifi, S. Otta, M. A. Nabian, M. Stadler, C. Hundt, K. Azizzadenesheli, et al. Geometry-informed neural operator for large-scale 3d pdes. *Advances in Neural Information Processing Systems*, 36:35836–35854, 2023.

[32] Z. Li, H. Zheng, N. Kovachki, D. Jin, H. Chen, B. Liu, K. Azizzadenesheli, and A. Anandkumar. Physics-informed neural operator for learning partial differential equations. *ACM/JMS Journal of Data Science*, 1(3):1–27, 2024.

[33] B. Lin, Z. Mao, Z. Wang, and G. E. Karniadakis. Operator learning enhanced physics-informed neural networks for solving partial differential equations characterized by sharp solutions. *arXiv preprint arXiv:2310.19590*, 2023.

[34] Z. Liu, F. Ni, and H. Zhou. Modeling and simulating for tshd's swell compensator by adams. *Journal of Ocean University of China*, 6(1):95–99, 2007.

[35] L. Lu, P. Jin, and G. E. Karniadakis. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *arXiv preprint arXiv:1910.03193*, 2019.

[36] R. Mattey and S. Ghosh. A novel sequential method to train physics informed neural networks for allen cahn and cahn hilliard equations. *Computer Methods in Applied Mechanics and Engineering*, 390:114474, 2022.

[37] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[38] N. Mohajerin and S. L. Waslander. Multistep prediction of dynamic systems with recurrent neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 30:3370–3383, 2019.

[39] A. Mollaali, İ Şahin, I. Raza, C. Moya, G. Paniagua, and G. Lin. A physics-guided bi-fidelity fourier-featured operator learning framework for predicting time evolution of drag and lift coefficients. *Fluids*, 8:323, 2023.

[40] M. Narkhede, P. Bartakke, and M. S. Sutaone. A review on weight initialization strategies for neural networks. *Artificial Intelligence Review*, 55:291–322, 2021.

[41] A.J. Nobel and A.M. Talmon. Measurements of the stagnation pressure in the center of a cavitating jet. *Experiments in Fluids*, 52(2):403 – 415, 2012.

[42] C. Oberwinkler and M. Stundner. From real-time data to production optimization. *SPE Production Amp; Facilities*, 20:229–239, 2005.

[43] R. L. Panton. *Incompressible flow*. John Wiley & Sons, 2024.

[44] R. Rai and C. K. Sahu. Driven by data or derived through physics? a review of hybrid physics guided machine learning techniques with cyber-physical system (cps) focus. *IEEE Access*, 8:71050–71073, 2020.

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

[46] A. Rashidi, H. R. Nejad, and M. Maghiar. Productivity estimation of bulldozers using generalized linear mixed models. *KSCE Journal of Civil Engineering*, 18:1580–1589, 2014.

[47] M. Sadoughi and C. Hu. A physics-based deep learning approach for fault diagnosis of rotating machinery. In *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, pages 5919–5923, 2018.

[48] M. Sadoughi and C. Hu. Physics-based convolutional neural network for fault diagnosis of rolling element bearings. *IEEE Sensors Journal*, 19(11):4181–4192, 2019.

[49] A. Savitzky and M. Golay. Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry*, 36:1627–1639, 1964.

[50] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[51] J. Shen, C. Chang, S. Wu, C. Hsu, and H. Lien. Real-time correction of water stage forecast using combination of forecasted errors by time series models and kalman filter method. *Stochastic Environmental Research and Risk Assessment*, 29:1903–1920, 2015.

[52] Z. Shen, Y. Zhang, J. Lu, J. Xu, and G. Xiao. A novel time series forecasting model with deep learning. *Neurocomputing*, 396:302–313, 2020.

[53] A. Shrestha and A. Mahmood. Review of deep learning algorithms and architectures. *IEEE Access*, 7:53040–53065, 2019.

[54] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 12 2018.

[55] J. Sun, C. Yan, and J. Wen. Intelligent bearing fault diagnosis method combining compressed data acquisition and deep learning. *IEEE Transactions on Instrumentation and Measurement*, 67(1):185–195, 2018.

[56] J. Tang and Q. Wang. Online fault diagnosis and prevention expert system for dredgers. *Expert Systems With Applications*, 34:511–521, 2008.

[57] J. van het Hof. *Advanced Training in Execution of Hydraulic Engineering Works. Printing*. Printing Publishing Protocol B.V., 2010.

[58] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[59] W. J. Vlasblom. *Trailing Suction Hopper Dredger*. 2005.

[60] S. Wang, H. Wang, and P. Perdikaris. Learning the solution operator of parametric partial differential equations with physics-informed deeponets. *Science Advances*, 7, 2021.

[61] S. Wang, H. Wang, and P. Perdikaris. Improved architectures and training algorithms for deep operator networks. *Journal of Scientific Computing*, 92, 2022.

[62] Z. Wei, W. Chen, and P. Stinis. Efficient transformer-inspired variants of physics-informed deep operator networks. *arXiv preprint arXiv:2509.01679*, 2025.

[63] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78:1550–1560, 1990.

[64] J. Willard, X. Jia, S. Xu, M. Steinbach, and V. Kumar. Integrating scientific knowledge with machine learning for engineering and environmental systems. *ACM Comput. Surv.*, 55(4), November 2022.

[65] J. Xing. Estimation and control of the overflow loss of trailing suction hopper dredger based on particle filter. Master's dissertation, School of Naval Architecture Ocean Engineering, Jiangsu University of Science and Technology, 2012. In Chinese.

[66] W. Xu, Y. Lu, and L. Wang. Transfer learning enhanced deeponet for long-time prediction of evolution equations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 10629–10636, 2023.

[67] S. Yang, Y. Lee, and N. Kang. Physics-guided multi-fidelity deeponet for data-efficient flow field prediction. *arXiv preprint arXiv:2503.17941*, 2025.

[68] W. Yang, W. Wang, X. Zhang, S. Sun, and Q. Liao. Lightweight feature fusion network for single image super-resolution. *IEEE Signal Processing Letters*, 26(4):538–542, 2019.

[69] B. Zamanlooy and M. Mirhassani. Efficient vlsi implementation of neural networks with hyperbolic tangent activation function. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(1):39–48, 2013.

[70] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[71] G. Zhang, Y. Duan, G. Pan, Q. Chen, H. Yang, and Z. Zhang. Parameter identification for partial differential equations with spatiotemporal varying coefficients. *arXiv preprint arXiv:2307.00035*, 2023.

[72] H. Q. Zhang and Y. Yan. A wavelet-based approach to abrupt fault detection and diagnosis of sensors. *IEEE Transactions on Instrumentation and Measurement*, 50(5):1389–1396, 2001.

[73] W. Zhang, Y. Jiang, J. Dong, X. Song, R. Pang, B. Guoan, and H. Yu. A deep learning method for real-time bias correction of wind field forecasts in the western north pacific. *Atmospheric Research*, 284:106586, 2023.

Appendices

A Testing Relations

In this section we go over how we chose the constants that made up the construction of our simulated dataset. We go through every relation, discussing the best fits. After this we discuss the importances of each individual variable.

Delta pressure draghead

The best fit that we receive based on the equation and the data is:

$$\theta = 0.71\Delta P_{head} + 2.44$$

The best fit is decent:

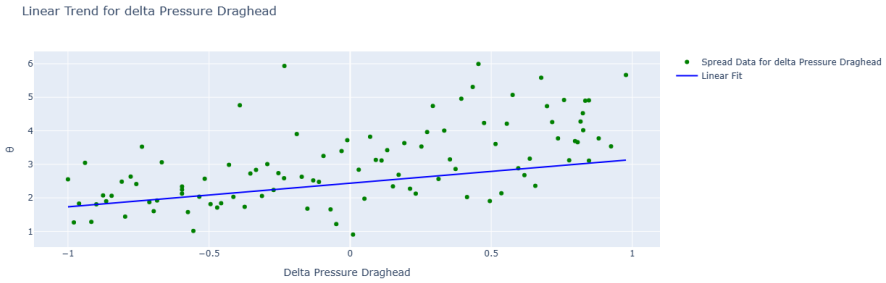


Figure 30: Linear fit. It seems that the data could more of an exponential or quadratic trend.

Mixture Density

The best fit that we received based on the equation (after adjusting the density data to be on the same time scale without delay) was:

$$\theta = \frac{1404}{\rho_m} + 1.52$$

The data spread also seems to be quite all over the place.

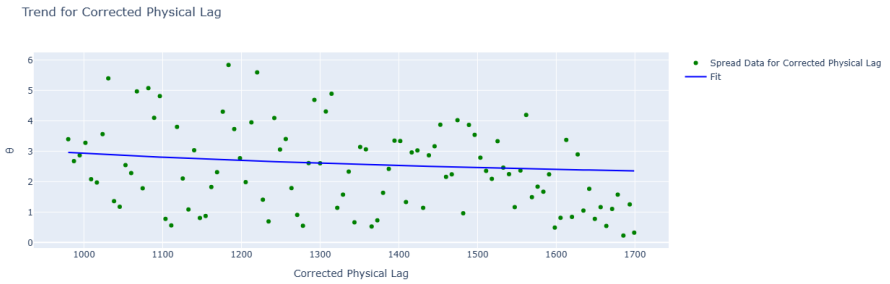


Figure 31: Linear fit for the corrected density. It seems that the data does not follow any trend and is all over the place.

Vacuum Pressure

The best fit that we receive based on the equation and the data is:

$$\theta = -1.08V + 1.84$$

The fit does not capture alot:

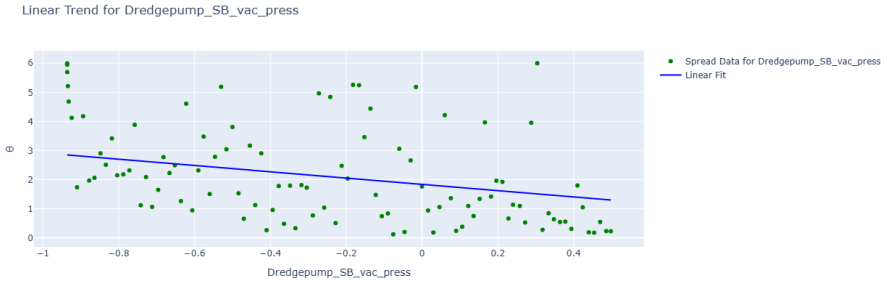


Figure 32: Linear fit. It seems that the data does not really follow any trend and is all over the place.

Mixture velocity

The best fit that we receive based on the equation and the data is:

$$\theta = \frac{4.18}{v^2} + 2.24$$

With the fit capturing the data well:

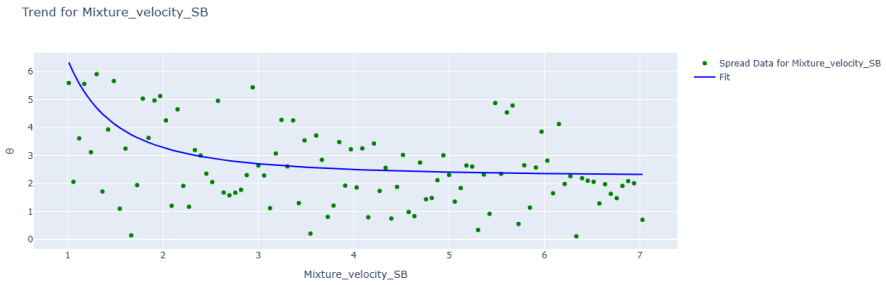


Figure 33: Inverse quadratic fit. It seems that the fit follows the data decently well.

Lower Pipe Angle

The best fit that we receive based on the equation and the data is:

$$\theta = -0.01\delta + 2.84$$

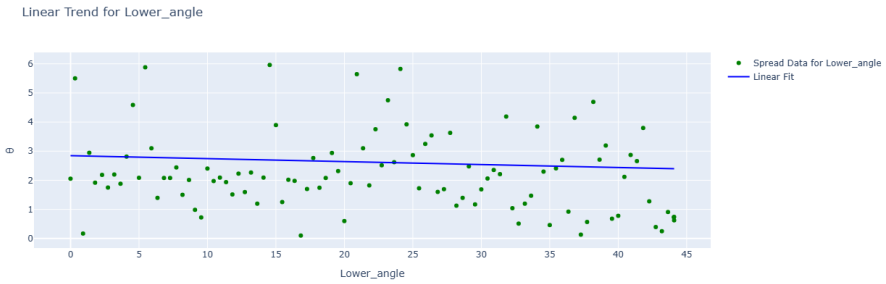


Figure 34: Linear trend. Data seems to not have any vertical movements but is steady at one point.

Jet pumps

The best fit that we receive based on the equation and the data is:

$$\theta = \frac{2.54}{P_j} + 2.31$$

The best fitting line does not seem to capture the data that well:

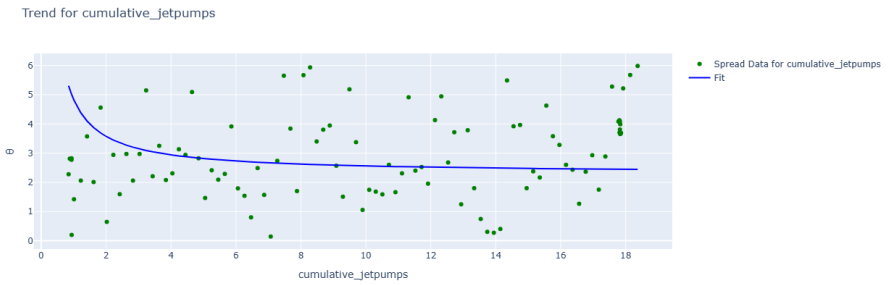


Figure 35: Data does not seems to follow the trend that well, maybe a logarithmic fit would work better.

Swellcompensator

The best fit that we receive based on the equation and the data is:

$$\theta = 0.02S_c + 1.47$$

The best fitting line does not seem to capture the data well at all:

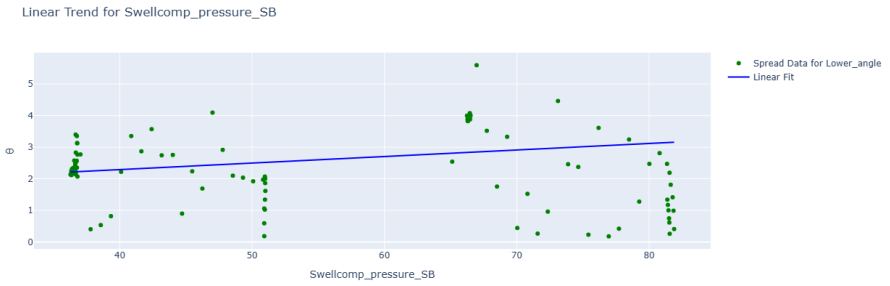


Figure 36: Data does not seems to follow the trend that well. There are also no data points in between 50 and 70 on the x axis.

Draghead Depth

The best fit that we receive based on the equation and the data is:

$$\theta = -0.04D + 3.31$$

The best fitting line does not seem to capture the data that well:

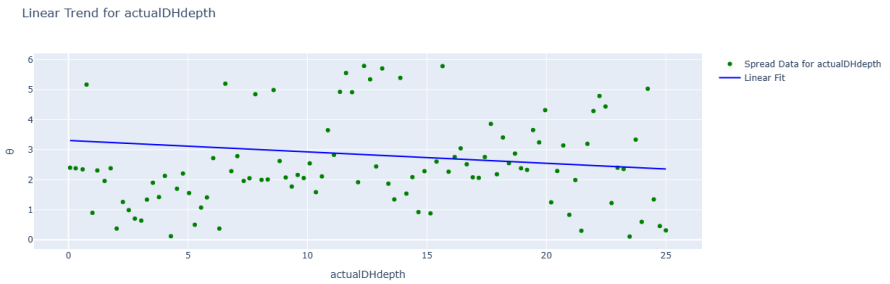


Figure 37: Data does not seems to follow the trend that well. It seems random.

Weighted Sum

Constructing a sequential least squares optimization problem (with L_2 regularization to penalize large weights in only one variable) to find out the weights, we end up with:

$$\theta_{Pred} = 0.08\theta_{\Delta P_{Head}} + 0.22\theta_{Vac} + 0.09\theta_{Density} + 0.13\theta_{Vel} +$$
$$0.07\theta_{Angle} + 0.11\theta_{Jet\ pumps} + 0.20\theta_{SwellComp} + 0.09\theta_{DHDepth}$$

It is important to keep in mind that many of these values could be inaccurate, This is because a change in the θ is caused by a combination of many different factors, and a fluctuation of θ is definitely not caused by only one variable. Nevertheless, we can build upon these equations and use them as a guideline to

simulate our own dataset.

A.1 Final data set

Based on the research done on the different variables and collaboration with experts, we construct our final dataset in the following way.

$$\theta_{\Delta P_{Head}} = 0.80\Delta P_{head} + 2.00$$

$$\theta_{Vac} = 2.00 - 1.50V$$

$$\theta_{Density} = \frac{1400}{\rho_m} + 1.50$$

$$\theta_{Vel} = \frac{4.00}{v^2} + 2.00$$

$$\theta_{Angle} = 0.02\delta + 2.50$$

$$\theta_{Jet\ pumps} = \frac{2.50}{P_j} + 2.30$$

$$\theta_{S_c} = -0.04S_c + 4.50$$

$$\theta_{DHDepth} = 0.05D + 2.00$$

$$\theta_{Sim} = 0.28\theta_{\Delta P_{Head}} + 0.18\theta_{Vac} + 0.13\theta_{S_c} + 0.12\theta_{Vel} +$$
$$0.11\theta_{Density} + 0.08\theta_{DHDepth} + 0.05\theta_{Jet\ pumps} + 0.05\theta_{Angle}$$

Explanations

The struggle is to find a balance between what we know about the process, and what the results of the testing are. We cannot rely too much on the test because we know they are not completely accurate. At the same time we can also not solely rely on our knowledge because then we would not be able to set up any relations.

Regarding the ΔP_{Head} relation, we know this is pretty much the defining factor for θ. Since this is the factor that we are trying to estimate the best, we increased the correlation coefficient and in turn reduced the intercept.

For the vacuum relation, we decide to increase the correlation slightly and round the intercept. This is because at more powerful vacuums, we know for

certain that θ gets impacted by it. Note that there is a negative sign in front of the vacuum, this is because the sensor outputs the vacuum pressure as negative.

The density relation was also kept the same, other than some rounding. In the relation corresponding to the velocity, the data seemed to have a decent fit to the equation, so the only difference made was rounding the coefficients.

The relation with the lower pipe angle was changed because the theory tells us that an increase in the angle should cause an increase of θ. This is why the sign was flipped to account for this. The reason why the data did not pick up on this is that this effect is very situational. If the draghead is deep in the seabed and the angle is lowered, θ could increase because the opening of the draghead would be flat on the seabed.

We keep the jet pump relation relatively the same as well, only rounding the values.

For the relation with the draghead depth, we also change it. This is because the theory is not reflected back in the relation. We know that the relation is correct due to its adoption from the vacuum equation. The same logic is applied to the swellcompensator, whereby we also altered the sign of the coefficient based on the prior knowledge. We take the gradients of both lines to be similar to the one we explored nevertheless.

We also altered the weights of each variable towards the final θ. We can deduce a rank of importances based on prior knowledge. We took the change in pressure draghead as the most important variable because this signal determines the magnitude of θ. The velocity, vacuum and swellcompensator follow because they are deemed quite important from testing (the vacuum and velocity also being influential through the vacuum equation). The draghead depth and density are also relevant through the vacuum equation. Finally, the least influential variables are the jet pumps and the lower pipe angle. This makes sense because they do not directly influence θ through the vacuum equation. Through testing they were also deemed to not be very important.

B Classical DeepONet and MIONet Results

We inspect the results of both the MIONet, as well as the classical DeepONet. Both models were trained with the same hyperparameters used to train the modified DeepONet. The MIONet functions the same as the modified net, except that it operates with a global trunk and no interactions. For the classical DeepONet, all inputs are concatenated into one input function. The DeepONet also makes use of a global trunk. Due to this global trunk, the predictions for different time windows are different. We compare the predictions by both averaging the time steps, as well as using the first predictions per window.

For the classical DeepONet, we inspect the performance for a training and test trip. It seems to predict a global mean across the trip, both for the test, as well the training trip. Both predictions are straight horizontal lines. We only present the results when averaging the results in time windows, since the first predictions look identical.

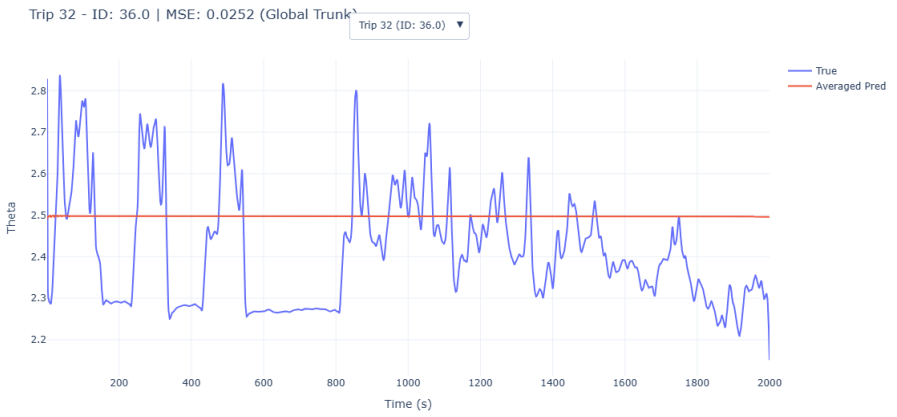


Figure 38: Classical DeepONet performance on a training trip (averaged predictions).

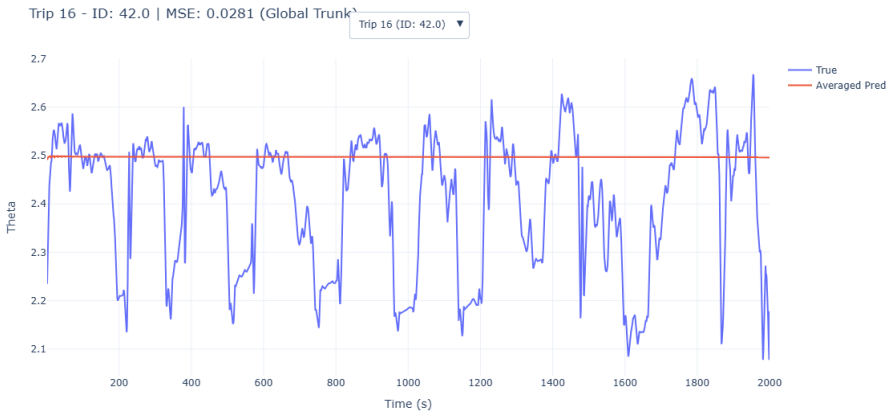


Figure 39: Classical DeepONet performance on a testing trip (averaged predictions). It still predicts a straight line.

The likely problem as to why the model performs poorly is due to two reasons. First concatenating all input functions into one, does not represent the data well at all. Especially since we have eight input functions to work with. That is, for m input functions, the sole input function for the DeepONet was constructed as: $F = [f_1(t_{s_k}, \ldots, t_{e_k}), \ldots, f_m(t_{s_k}, \ldots t_{e_k})]$. This makes it challenging for the DeepONet to learn temporal patterns. Perhaps multiplying or adding all input functions would yield stronger results. Then, it also works with a global trunk. It tries to encode all the input windows with the same representative basis functions, causing the predictions for individual windows to smoothen out, this can be seen in a training window in fig. 40.

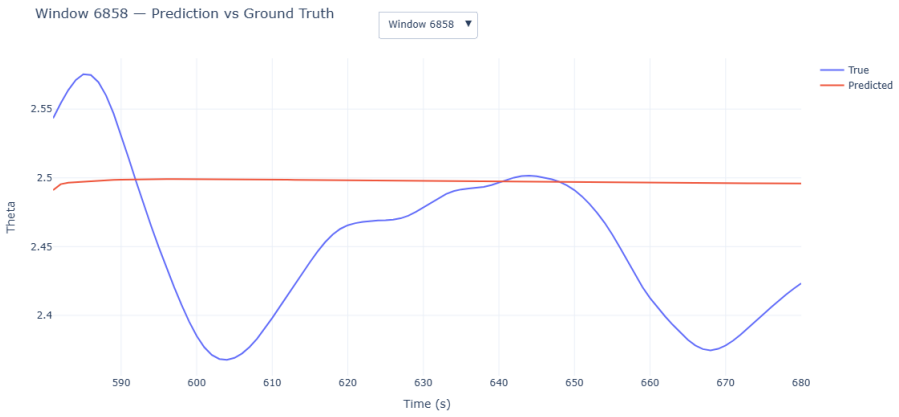
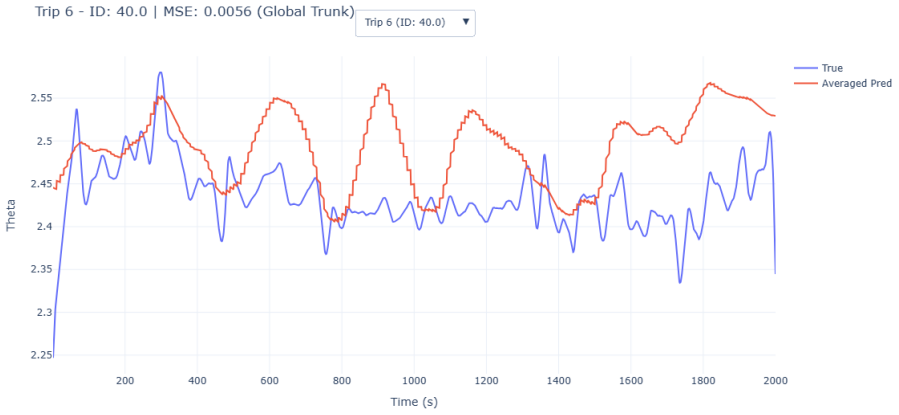
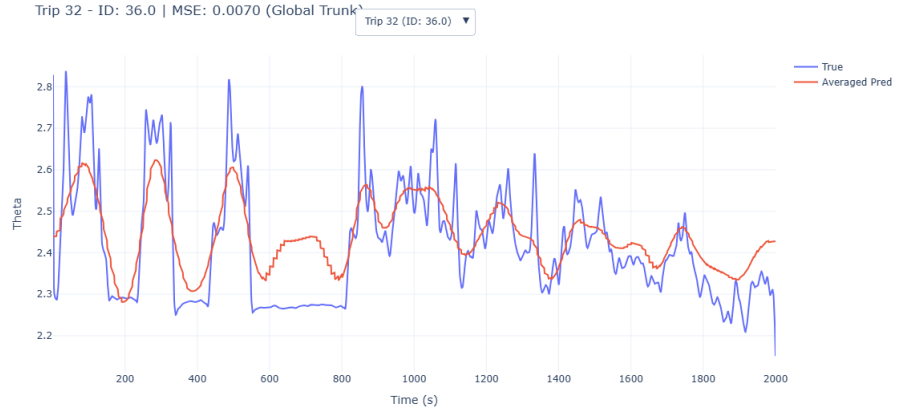


Figure 40: Training window for the classical DeepONet, this (nearly) straight line pattern is repeated for all of the windows it trains on.

Looking at the MIONet, we see that it struggles to capture intricate patterns. It seems to act like it smooths out the predictions, similar to a filter. It does not capture any meaningful spikes. The overall trend is captured poorly.



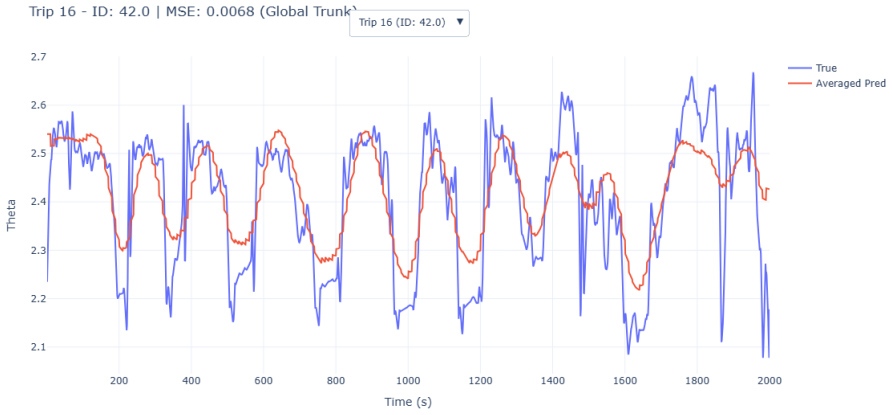
(a) Predictions for the easier trip. Model expects the trip to fluctuate more even though that was not that case.



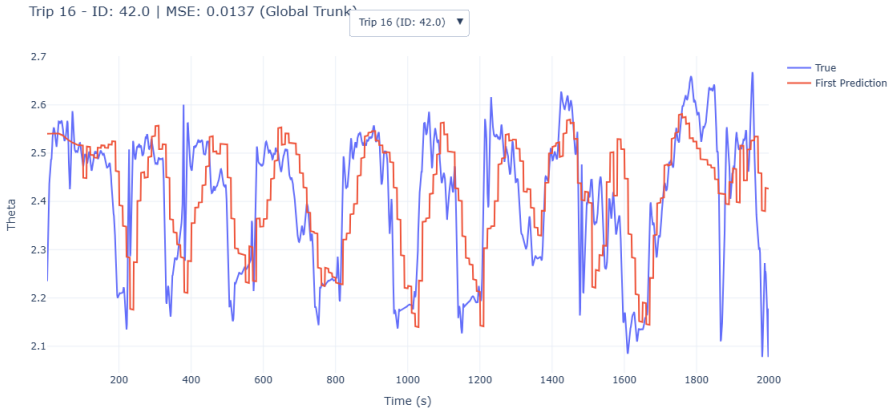
(b) Predictions for the trip with more fluctuations. Blocky behaviour is present but overall pattern is captured throughout the trip.

Figure 41: Comparison of different training trips from the MIONet.

Now when looking at the performance on a testing trip, we compare how the averaged predictions look like together with the first predictions.



(a) Predictions for a test set with the values averaged from every window.



(b) Same testing trip but by taking the first prediction per time window.

Figure 42: Comparison of methodologies on a testing trip from the MIONet. We observe that the first prediction graph exhibits a more blocky behaviour, whereas the averaged predictions are smoother.

We also showcase a training window for the MIONet:

Figure 43: Training window on the MIONet. Instead of a straight line imitating a global mean in fig. 40, this behaviour showcases a sort of smoothing on the window, not capturing any fluctuations.

We see that the prediction windows also exhibit block-type behaviour. Similar to the classic DeepONet, this is because a global trunk is used. Using the global trunk does not allow for enough representation for the basis functions of the time, meaning that it settles for an average effect, which causes the smoothing. We inspect the training for a single window in fig. 43, and see that this is also smoothed, even though you expect the model to try and overfit on the training windows. Here we see why using a local trunk was a better idea, and why it worked out better. We summarize the results on simulated data in tables 14 and 15.

Table 14: Performance metrics for training trips.

Architecture	**Training Time (hours)**	**RMSE**	**MAE**	$\mathbf{R^2}$
ModNet	8	0.0124	0.0106	0.9589
Corrected ModNet	8	0.0066	0.0032	0.9886
Baseline	0	0.0577	0.0354	0.3082
MIONet(averaged)	3	0.0460	0.0328	0.2587
MIONet(first)	3	0.0538	0.0382	0.1320
Classical DeepONet (avg)	1	0.1039	0.0883	-1.4724
Classical DeepONet (first)	1	0.1041	0.0885	-1.5014

Table 15: Evaluation metrics across architectures on the testing trips.

Architecture	**RMSE**	**MAE**	$\mathbf{R^2}$
ModNet	0.0148	0.0115	0.9416
Corrected ModNet	0.0094	0.0036	0.9745
Baseline	0.0592	0.0359	0.2647
MIONet(avg)	0.0665	0.0487	-0.0449
MIONet(first)	0.0759	0.0552	-0.2646
Classical DeepONet(avg)	0.1003	0.0855	-1.4015
Classical DeepONet(first)	0.1007	0.0859	-1.4521

We can see that the classical DeepONet is the worst performing network out of all, with the scores for the averaged and first predictions being equally bad. We also see that the classical DeepONet performs better on the test set rather than the training set. As we have seen in the previous figures, this network predicts mostly a straight line for every trip. This straight line, on average, fits the testing trips better than the training trips. This may just be because of the choice of training and testing trips. The averaged predictions for the MIONet narrowly outperforms the baseline, with the first predictions not being

much better than the baseline. The advantages that the classical DeepONet and MIONet have, is that the training time is significantly shorter. The baseline on the other hand, does not take any time to train at all).

C Results For Every Trip

Here we present every trip used in the testing set from the simulated data. In all graphs the actuals are plotted together with the predicted and corrected values. In total there are 17 testing trips. It is impressive to see how the model manages to capture different types of behaviour extremely well.

Additionally, we list the total variation scores for every trip (train and test) in table 16:

Table 16: Total variation per trip for `theta_sim`

Trip ID	**Total variation**	**Set**
1.0	3.579897	Training
2.0	6.308544	Training
5.0	4.386708	Training
6.0	8.637192	Training
7.0	8.473776	Training
9.0	9.056531	Training
10.0	7.799481	Training
12.0	7.965652	Training
14.0	4.706631	Training
15.0	3.062204	Training
17.0	2.826278	Training
18.0	3.139688	Training
19.0	4.519754	Training
21.0	2.686614	Training
23.0	3.041902	Training

Table 16 (continued)

Trip ID	Total variation	Set
26.0	3.561921	Training
27.0	2.735935	Training
30.0	7.173509	Training
31.0	9.552793	Training
33.0	7.821018	Training
34.0	7.893401	Training
35.0	9.225170	Training
36.0	13.301205	Training
37.0	12.710077	Training
38.0	4.273602	Training
40.0	3.239353	Training
41.0	3.841955	Training
45.0	3.192325	Training
46.0	1.909604	Training
49.0	3.460514	Training
51.0	3.396717	Training
52.0	3.266302	Training
53.0	3.679815	Training
54.0	4.656152	Training
55.0	4.184005	Training
57.0	4.021062	Training
58.0	5.303052	Training
59.0	5.671699	Training
3.0	8.412444	Testing
8.0	6.982695	Testing
11.0	5.538218	Testing
16.0	4.149214	Testing
20.0	2.604748	Testing

Table 16 (continued)

Trip ID	Total variation	Set
22.0	3.120981	Testing
24.0	2.833151	Testing
25.0	4.075747	Testing
32.0	10.469907	Testing
39.0	5.407156	Testing
42.0	18.779810	Testing
44.0	2.450384	Testing
47.0	3.555539	Testing
48.0	5.108872	Testing
50.0	4.310129	Testing
56.0	5.259841	Testing
60.0	5.215833	Testing

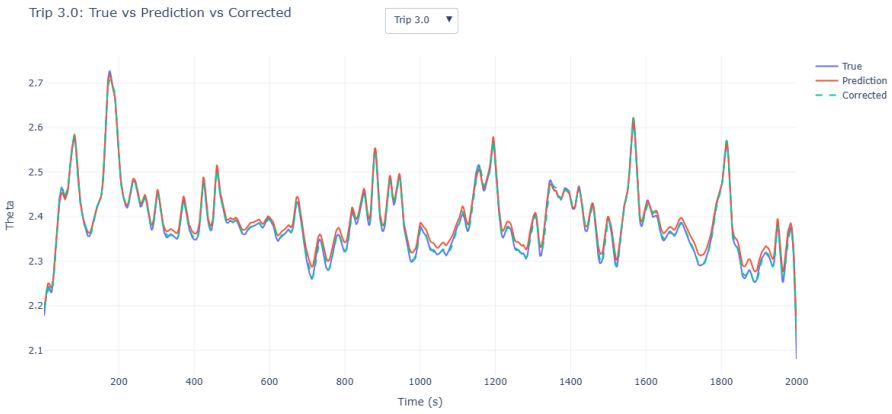


Figure 44: Actuals, predictions and corrections for trip with ID 3.

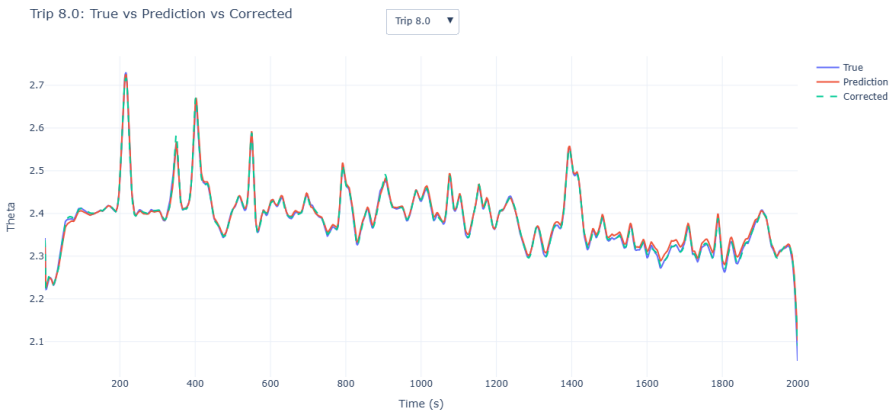


Figure 45: Actuals, predictions and corrections for trip with ID 8.

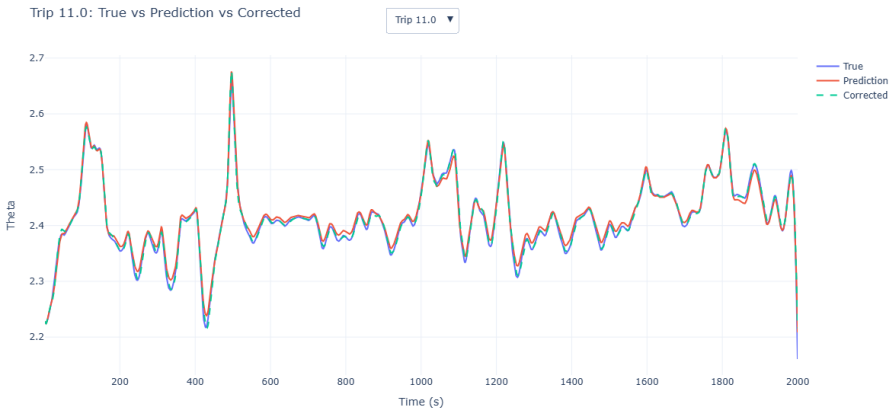


Figure 46: Actuals, predictions and corrections for trip with ID 11.

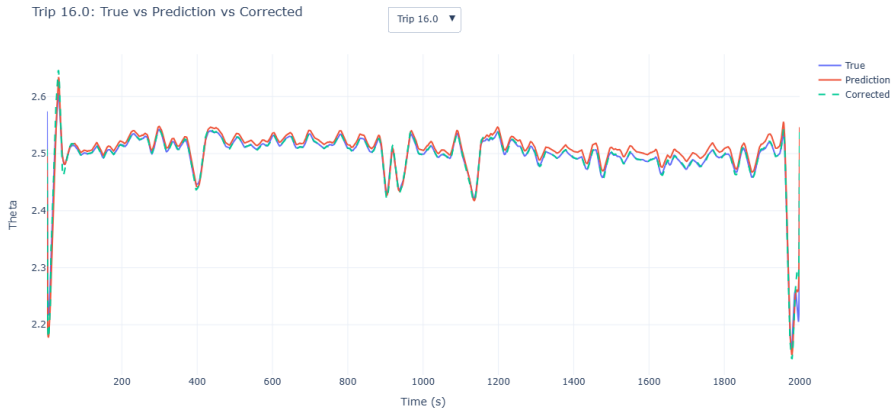


Figure 47: Actuals, predictions and corrections for trip with ID 16.

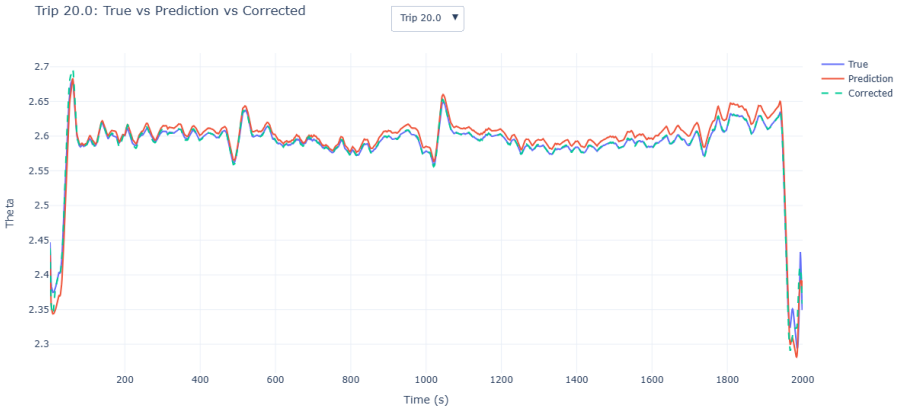


Figure 48: Actuals, predictions and corrections for trip with ID 20.

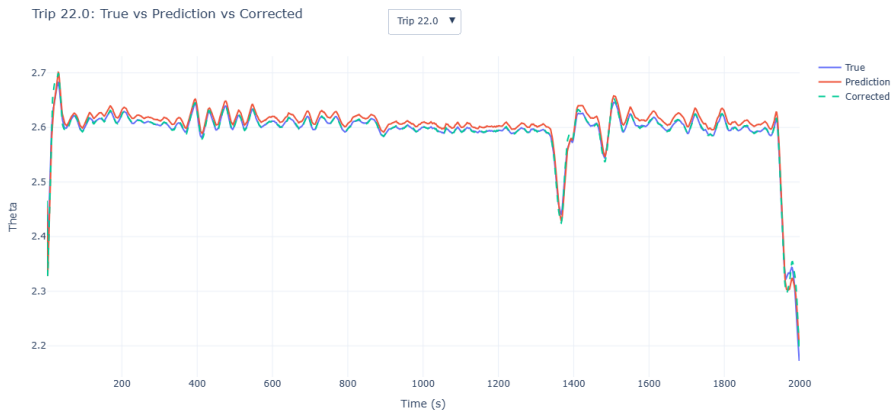


Figure 49: Actuals, predictions and corrections for trip with ID 22.

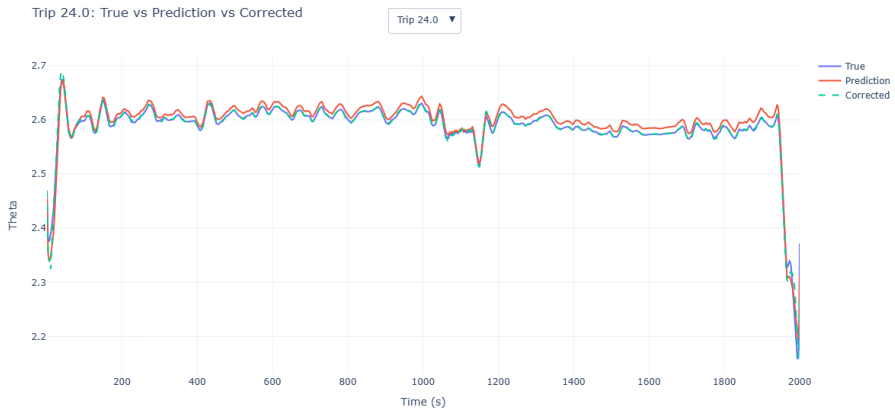


Figure 50: Actuals, predictions and corrections for trip with ID 24.

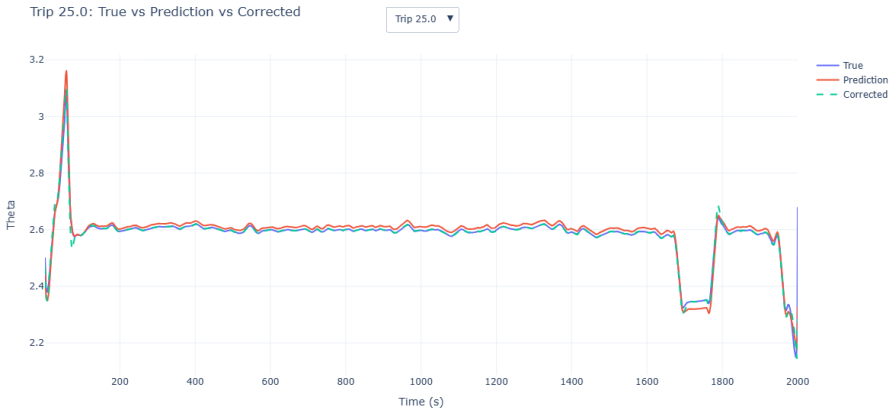


Figure 51: Actuals, predictions and corrections for trip with ID 25.

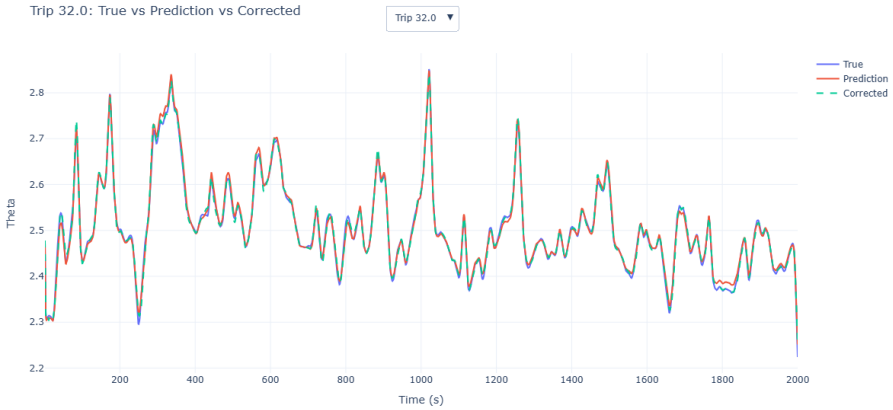


Figure 52: Actuals, predictions and corrections for trip with ID 32.

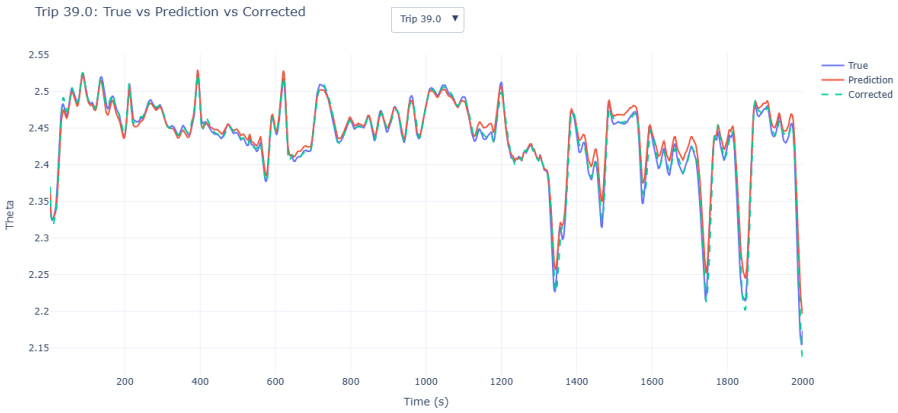


Figure 53: Actuals, predictions and corrections for trip with ID 39.

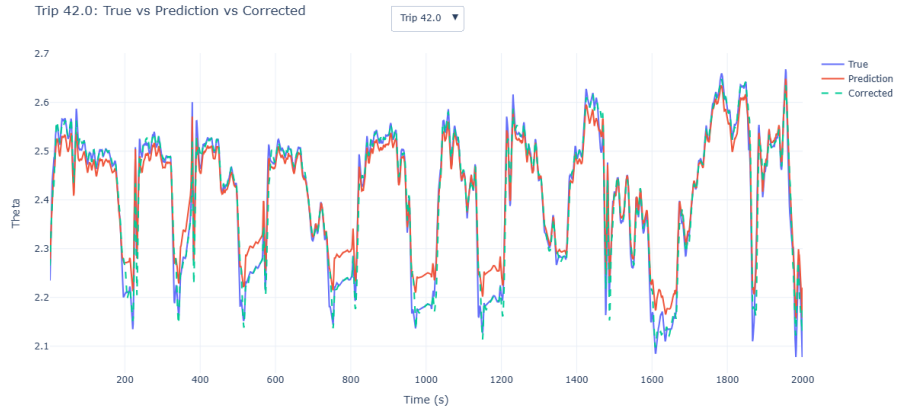


Figure 54: Actuals, predictions and corrections for trip with ID 42.

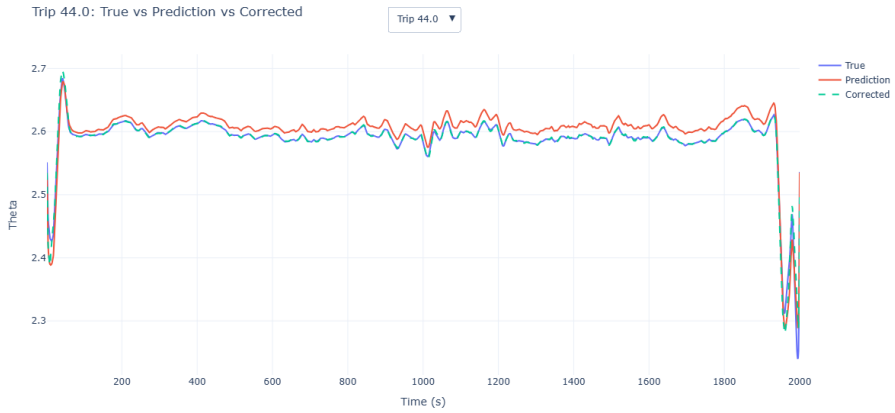


Figure 55: Actuals, predictions and corrections for trip with ID 44.

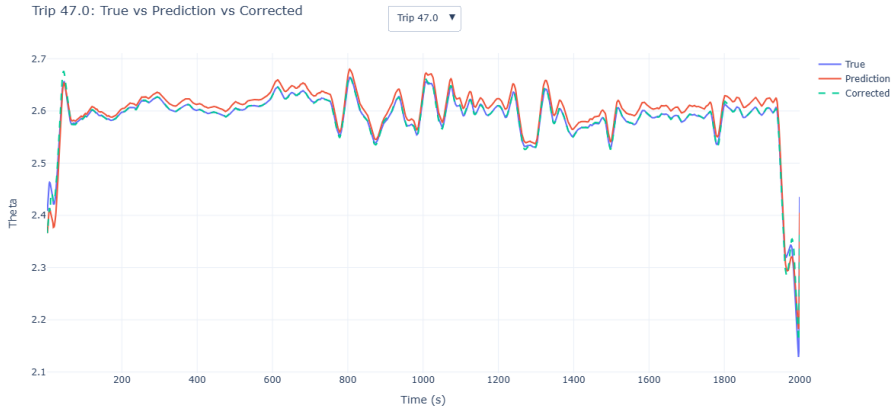


Figure 56: Actuals, predictions and corrections for trip with ID 47.

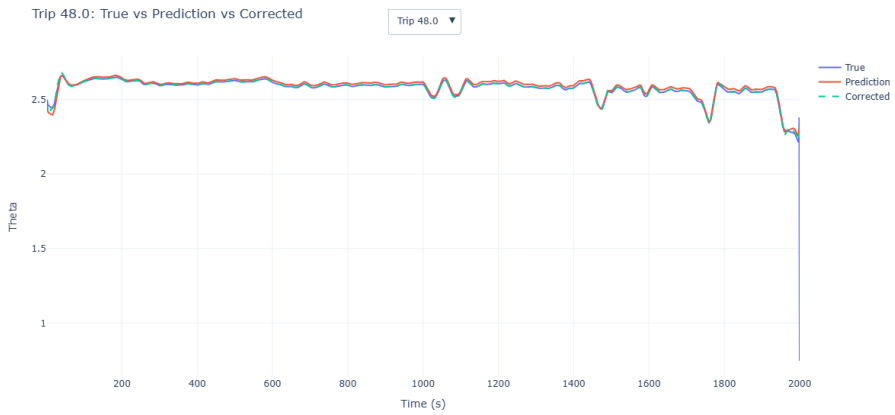


Figure 57: Actuals, predictions and corrections for trip with ID 48.

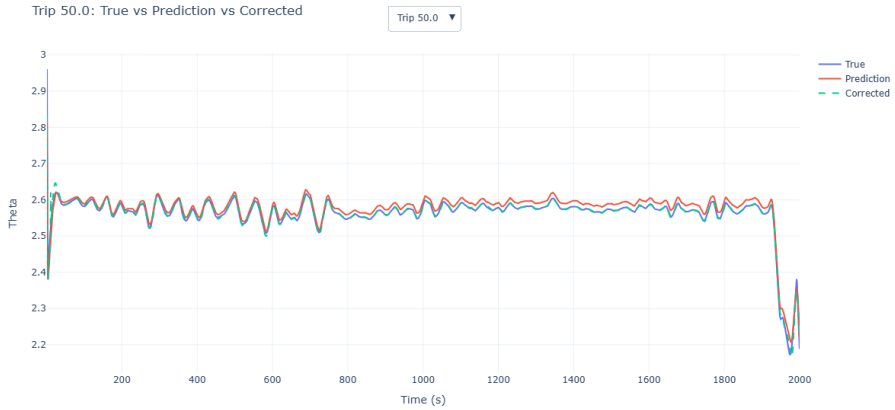


Figure 58: Actuals, predictions and corrections for trip with ID 50.

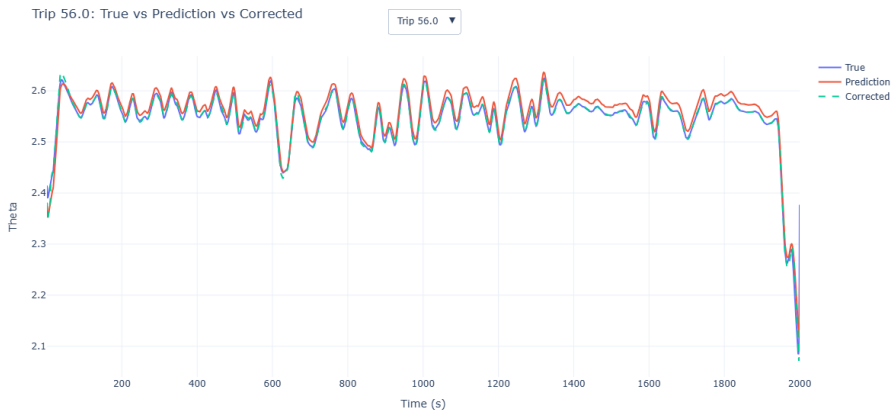


Figure 59: Actuals, predictions and corrections for trip with ID 56.

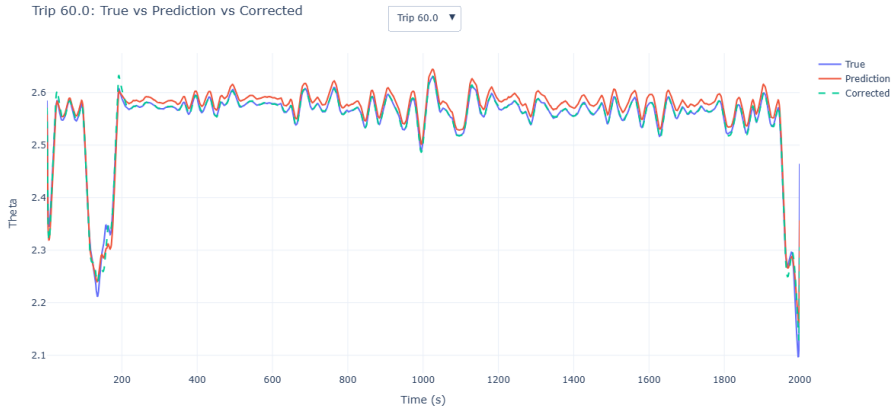


Figure 60: Actuals, predictions and corrections for trip with ID 60.