



Università degli Studi di Padova

DEPARTMENT OF MATHEMATICS “TULLIO LEVI-CIVITA”

Master Degree in Mathematics

Goal Oriented Operator Networks

Supervisor:
Prof. Fabio Marcuzzi

Candidate: Enrico Caregnato
Student ID number: 2097351

Co-Supervisors:
Dott.ssa Laura Rinaldi
Dott. Marco Dell’Orto

Academic Year 2024-2025

11/04/2025

A Valentina, ai miei genitori

Contents

ABSTRACT	vii
1 INTRODUCTION TO NEURAL NETWORKS	1
1.1 Multilayer perceptron architecture	1
1.2 Training, validation and testing of neural networks	3
1.3 Generalizability and regularization	4
1.4 Gradient descent	5
1.5 An advanced optimization algorithm	7
1.6 Calculating gradients using back propagation	9
2 OPERATOR NETWORKS	13
2.1 Physics-Informed Neural Networks	13
2.1.1 Parametrized PDEs	14
2.2 DeepONet architecture	15
2.3 Training DeepONets	17
2.4 Error analysis for DeepONet	19
3 VARIATIONALLY MIMETIC OPERATOR NETWORKS	21
3.1 PDE model and discretization	21
3.1.1 Space-time weak formulation	22
3.1.2 Discretization of the problem	23
3.2 VarMiON architecture	25
4 GOAL ORIENTED VARMiON	29
4.1 Numerical integration	29
4.2 Loss function	32
4.3 Numerical experiments	35
4.3.1 Dataset	35

4.3.2	Relative error	36
4.3.3	Analysis of predictions	36
5	KALMAN FILTER	41
5.1	Kalman Filter equations: general case	41
5.2	Kalman Filter equations: heat equation	42
5.3	Numerical experiments	46
5.3.1	Case: $\sigma_R^2 = 10^{-2}$, $\sigma_P^2 = 10^{-2}$	46
5.3.2	Case: $\sigma_R^2 = 10^{-2}$, $\sigma_P^2 = 10^{-6}$	49
5.3.3	Case: $\sigma_R^2 = 10^{-6}$, $\sigma_P^2 = 10^{-2}$	51
5.3.4	Case: $\sigma_R^2 = 10^{-6}$, $\sigma_P^2 = 10^{-6}$	54
6	CONCLUSIONS AND FUTURE DEVELOPMENTS	57
	BIBLIOGRAPHY	59

Abstract

In recent years, operator networks have played a central role in solving complex problems in various scientific and technological fields. This thesis focuses on *Goal Oriented Operator Networks*, with particular emphasis on their application in solving partial differential equations (PDEs), specifically the heat equation.

After a general introduction to neural network architectures, we present an overview of Physics-Informed Neural Networks (PINNs) and Deep Operator Networks (DeepONets), by analyzing their capabilities and limitations.

Next, we introduce the concept of Variationally Mimetic Operator Networks (VarMiON), an advanced architecture inspired by the variational formulation of any given PDE. We analyze its learning and approximation capabilities, in the case of the heat equation, highlighting the contribution of goal oriented loss functions. Specifically, we propose a modification of VarMiON to improve network accuracy in specific regions of the domain.

Finally, the thesis explores the application of operator networks in the Kalman filter, combining VarMiON with a well-known estimation technique to enhance state prediction in dynamic systems. The numerical results demonstrate that the proposed approach enables greater accuracy in predicting heat equation solutions, particularly in predefined areas of interest.

Chapter 1

Introduction to neural networks

In this first chapter, we will introduce the simplest neural network architecture, known as *multilayer perceptron* (MLP), and we will discuss the main steps involved in the network learning process [8].

1.1 Multilayer perceptron architecture

We consider the problem of approximating a function $\mathbf{f} : \mathbf{x} \in \mathbb{R}^d \mapsto \mathbf{y} \in \mathbb{R}^D$ using an MLP that we denote \mathcal{F} . The MLP is the simplest network architecture available, it consists of many units, called *artificial neurons*, stacked in a number of consecutive layers. The zeroth layer of \mathcal{F} is called the *source layer* and is only responsible for providing an input to the network; the last layer of \mathcal{F} is known as the *output layer*, which outputs the network prediction; every other layer in between is known as a *hidden layer*. A schematic of an MLP with 2 hidden layers is shown in Figure 1.1.

To understand how an MLP works, we consider a network with L hidden layers, with the *width* of layer (l) denoted as H_l for $l = 0, 1, \dots, L + 1$. Then, for consistency with the function f that we are trying to approximate, we must have $H_0 = d$ and $H_{L+1} = D$. We also denote the output vector for l -th layer by $x^{(l)} \in \mathbb{R}^{H_l}$, which will serve as the input to the next layer. Lastly, we set $x^{(0)} = x \in \mathbb{R}^d$ which will be the input signal provided by the input layer. What happens in the network is that in each layer l , $1 \leq l \leq L + 1$, the i -th neuron performs an affine transformation on the input received from the previous one, followed by a non-linear transformation:

$$x_i^{(l)} = \sigma(W_{ij}^{(l)} x_j^{(l-1)} + b_i^{(l)}), \quad 1 \leq i \leq H_l, \quad 1 \leq j \leq H_{(l-1)} \quad (1.1)$$

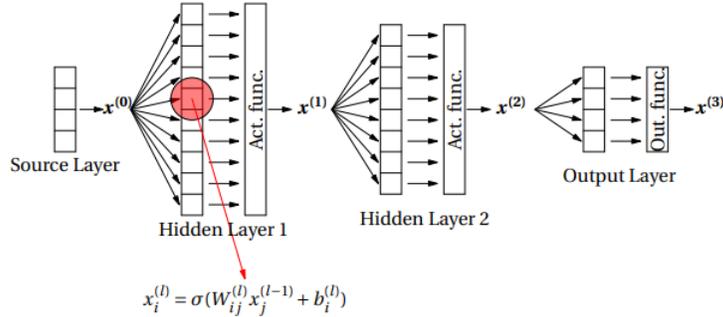


Figure 1.1: MLP with two hidden layers

where $W_{ij}^{(l)}$ and $b_i^{(l)}$ are respectively known as the *weights* and *bias* associated with i -th neuron of layer l , while the function $\sigma(\cdot)$ is known as the *activation function*, and plays a crucial role in helping the network to represent non-linear complex functions. If we set $W^{(l)} \in \mathbb{R}^{H_{l-1} \times H_l}$ to be the weight matrix for layer l and $\mathbf{b}^{(l)} \in \mathbb{R}^{H_l}$ to be the bias vector for layer l , then we can re-write the action of the whole layer as

$$\mathbf{x}^{(l)} = \sigma(\mathcal{A}^{(l)}(\mathbf{x}^{(l-1)})), \quad \mathcal{A}^{(l)}(\mathbf{x}^{(l-1)}) = \mathbf{W}^{(l)}\mathbf{x}^{(l-1)} + \mathbf{b}^{(l)} \quad (1.2)$$

where the activation function is applied component-wise.

Thus, the action of the whole network $\mathcal{F} : \mathbb{R}^d \rightarrow \mathbb{R}^D$ can be mathematically seen as a composition of alternating affine transformations and component-wise activations:

$$\mathcal{F}(\mathbf{x}) = \mathcal{A}^{(L+1)} \circ \sigma \circ \mathcal{A}^{(L)} \circ \sigma \circ \dots \circ \sigma \mathcal{A}^{(1)}(\mathbf{x}). \quad (1.3)$$

The parameters of the network are all the weights and biases, which we will represent as

$$\boldsymbol{\theta} = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L+1} \in \mathbb{R}^{N_\theta}$$

where N_θ denotes the total number of parameters of the network.

Then we can see the network $\mathcal{F}(\mathbf{x}; \boldsymbol{\theta})$ as a family of parameterized functions, where $\boldsymbol{\theta}$ needs to be chosen appropriately so that the network approximates the target function $f(\mathbf{x})$ at the input \mathbf{x} .

What we notice is that the expressivity of the network strongly depends on the *depth* of the network, that is the number of computing layers, on the width and on the activation function that is used.

1.2 Training, validation and testing of neural networks

After having seen the architecture of MLPs, we discuss now how the parameters of these networks are set to approximate some target functions. We restrict in particular to the framework of supervised learning.

So, let us assume that we are given a dataset of pairwise samples $S = (x_i, y_i) : 1 \leq i \leq N$ corresponding to a target function $\mathbf{f} : \mathbf{x} \rightarrow \mathbf{y}$. We wish to approximate this function using the neural network

$$\mathcal{F}(\mathbf{x}; \boldsymbol{\theta}; \Theta)$$

where $\boldsymbol{\theta}$ are the network parameters defined before, while Θ corresponds to the *hyper-parameters* of the networks, such as depth, width, type of activation function, etc. The strategy for designing a robust network is divided into three steps.

1. *Training*: find the optimal values of $\boldsymbol{\theta}$ (for a fixed Θ);
2. *Validation*: find the optimal values of Θ ;
3. *Testing*: the performance of the network is tested on a set of unseen data.

To achieve these three objectives, it is common practice to divide the dataset S into three separate parts: a *training set* with N_{train} samples, a *validation set* with N_{val} samples and a *test set* with N_{test} samples, s.t. $N_{train} + N_{val} + N_{test} = N$. Usually these three sets take around 60%, 20% and 20% of the samples respectively.

This strategy is necessary since networks are heavily over-parameterized functions and the large number of degrees of freedom available to model the data can lead to over-fitting the data.

Now, let us explore in more detail how this split is applied during the three phases:

Training: Training the network requires solving the following optimization problem: Find $\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \Pi_{train}(\boldsymbol{\theta})$, where

$$\Pi_{train}(\boldsymbol{\theta}) = \frac{1}{N_{train}} \sum_{\substack{i=1 \\ (\mathbf{x}_i, \mathbf{y}_i) \in S_{train}}}^{N_{train}} \|\mathbf{y}_i - \mathcal{F}(\mathbf{x}_i; \boldsymbol{\theta}, \Theta)\|^2$$

for some fixed Θ . The function Π_{train} is known as the *loss function* and it is minimized using an appropriate gradient-based algorithm. In the example above, we have used the mean-squared loss function, but other types are available.

Validation: Validation of the network requires solving the following optimization problem: Find $\Theta^* = \arg \min_{\Theta} \Pi_{val}(\Theta)$, where

$$\Pi_{val}(\Theta) = \frac{1}{N_{val}} \sum_{\substack{i=1 \\ (\mathbf{x}_i, \mathbf{y}_i) \in S_{val}}}^{N_{val}} \|\mathbf{y}_i - \mathcal{F}(\mathbf{x}_i; \theta^*, \Theta)\|^2$$

In this case, the optimal Θ^* is obtained using a techniques such as grid search.

Testing: Once the "best" network is obtained, characterized by θ^* and Θ^* , it is evaluated on the test set S_{test} to estimate the performance of the network on data not used during the first two phases. The testing error is

$$\Pi_{test} = \frac{1}{N_{test}} \sum_{\substack{i=1 \\ (\mathbf{x}_i, \mathbf{y}_i) \in S_{test}}}^{N_{test}} \|\mathbf{y}_i - \mathcal{F}(\mathbf{x}_i; \theta^*, \Theta^*)\|^2$$

and it is also known as the (approximate) generalizing error of the network.

1.3 Generalizability and regularization

We discuss *generalizability* in the context of studying the ability of a trained network to perform well on data not used during training or validation steps.

This problem is non-trivial because if the network overfits the training data, then it is likely to produce inaccurate predictions on test data. Overfitting typically occurs because neural networks are almost always over-parametrized, which implies that the network model is highly non-linear. Consequently, the loss function $\Pi(\theta)$ (the subscript 'train' is omitted for brevity) often has many local minima and the challenge is to identify which minimum leads to a better generalization.

The primary technique used to address this issue is called *regularization*.

One of the simplest methods of regularization involves adding a penalty term to the loss function:

$$\Pi(\boldsymbol{\theta}) \rightarrow \Pi(\boldsymbol{\theta}) + \alpha \|\boldsymbol{\theta}\|, \quad \alpha \geq 0$$

where α is a regularization hyper-parameter, and $\|\cdot\|$ is a suitable norm of the network parameters. The penalty works encouraging the selection of minima that correspond to smaller value of the parameters $\boldsymbol{\theta}$. To understand why a smaller value of $\boldsymbol{\theta}$ would be a better choice because, consider the intermediate network output

$$x_1^{(1)} = \sigma(W_{1j}^{(1)} x_j^{(0)} + b_1^{(1)})$$

which gives

$$\frac{\partial x_1^{(1)}}{\partial x_1^{(0)}} = \sigma'(W_{1j}^{(1)} x_j^{(0)} + b_1^{(1)}) W_{11}^{(1)}.$$

Since this derivative scales with $W_{11}^{(1)}$, this implies that $|\frac{\partial \mathcal{F}(\mathbf{x})}{\partial x_1^{(0)}}|$ scales with $W_{11}^{(1)}$ as well. If $|W_{11}^{(1)}| \gg 1$, then the network would be very sensitive to even small changes in the input $x_1^{(0)}$, i.e., the network would be ill-posed.

1.4 Gradient descent

One of the main methods used to solve the minimization problem in the training step is the *gradient descent* (GD).

Consider the Taylor expansion of Π , centered on $\boldsymbol{\theta}_0$, with an increment of $\Delta\boldsymbol{\theta}$:

$$\Pi(\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta}) = \Pi(\boldsymbol{\theta}_0) + \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_0) \cdot \Delta\boldsymbol{\theta} + \frac{\partial^2 \Pi}{\partial \theta_i \partial \theta_j}(\hat{\boldsymbol{\theta}}) \Delta\theta_i \Delta\theta_j$$

for some $\hat{\boldsymbol{\theta}}$ in a small neighborhood of $\boldsymbol{\theta}_0$.

When $|\Delta\boldsymbol{\theta}|$ is small and assuming $\frac{\partial^2 \Pi}{\partial \theta_i \partial \theta_j}$ is bounded, we can neglect the second order term and just consider the approximation:

$$\Pi(\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta}) \approx \Pi(\boldsymbol{\theta}_0) + \frac{\partial \Pi}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}_0) \cdot \Delta\boldsymbol{\theta}.$$

Now, in order to lower the value of the loss function as much as possible compared to its evaluation at $\boldsymbol{\theta}_0$, we need to choose the step $\Delta\boldsymbol{\theta}$ in the opposite direction of the gradient, i.e.:

$$\Delta\boldsymbol{\theta} = -\eta \frac{\partial\Pi}{\partial\boldsymbol{\theta}}(\boldsymbol{\theta}_0)$$

with the step-size $\eta \geq 0$, also known as the *learning rate*. This is yet another hyper-parameter that we need to tune during the validation phase.

The procedure can then be summarized as follows.

1. Initialize $k = 0$ and $\boldsymbol{\theta}_0$
2. While $|\Pi(\boldsymbol{\theta}_k)| > \varepsilon_1$, do
 - (a) Evaluate $\frac{\partial\Pi}{\partial\boldsymbol{\theta}}(\boldsymbol{\theta}_k)$
 - (b) Update $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \frac{\partial\Pi}{\partial\boldsymbol{\theta}}(\boldsymbol{\theta}_k)$
 - (c) Increment $k = k + 1$

Convergence: Assume that $\Pi(\boldsymbol{\theta})$ is convex and differentiable, and its gradient is Lipschitz continuous with Lipschitz constant K . Then for $\eta \leq \frac{1}{K}$, the GD updates converge as

$$\|\boldsymbol{\theta}^* - \boldsymbol{\theta}_k\|_2 \leq \frac{C}{k}.$$

However, in most scenarios $\Pi(\boldsymbol{\theta})$ is not convex and, as said before, it may have more than one minima.

To understand the behavior of GD in that case, we consider the loss function for a scalar θ as shown in Figure 1.2, which has two valleys and we assume that the profile of $\Pi(\theta)$ in each valley can be approximated by a (centered) parabola:

$$\Pi(\theta) \approx \frac{1}{2}a\theta^2$$

where $a > 0$ is the curvature of each valley. Note that the curvature of the left valley is much smaller than the curvature of the right valley. Let us take a constant learning rate η and a starting value θ_0 in either of the valleys, then

$$\frac{\partial\Pi}{\partial\theta}(\theta_0) = a\theta_0$$

and the new point after a GD update will be $\theta_1 = \theta_0(1 - a\eta)$. Similarly, it is easy to see that all subsequent iterates are $\theta_{k+1} = \theta_k(1 - a\eta)$.

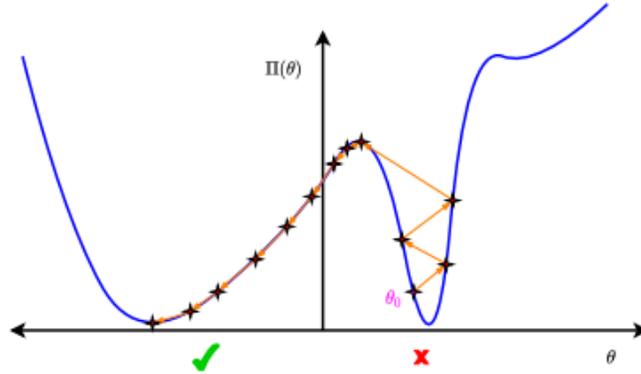


Figure 1.2: GD prefers flatter minima

Now for convergence, we need

$$\frac{\theta_{k+1}}{\theta_k} < 1 \implies |1 - a\eta| < 1.$$

Moreover, since $a > 0$ in the valleys, we also need the following condition on the learning rate:

$$-1 < 1 - a\eta \implies a\eta < 2.$$

In the end, if η is fixed, then for convergence the local curvature must satisfy $a < \frac{2}{\eta}$.

In other words, GD will prefer to converge to a minimum with a small curvature, that is the one in the left valley (see Figure 1.2). If the starting point is in the right valley, it may overshoot the minimum and rebound off the opposite wall repeatedly until the GD algorithm sends θ_k out of the valley. Once this occurs, the algorithm will move into the left valley, characterized by a lower curvature, and progressively approach its minimum.

To conclude, since it is clear that GD prefers flat minima, we also note that there is empirical evidence that the parameter values obtained at flat minima tend to generalize better, and therefore are to be preferred.

1.5 An advanced optimization algorithm

An issue with the GD method is that convergence to the minima may be quite slow if η is not chosen in a suitable way.

For example, consider the landscape of the objective function shown in Figure 1.3, which has sharper gradients along the $[\theta]_2$ direction compared to the $[\theta]_1$ direction, where $[\theta]_1$ and $[\theta]_2$ denote the components of θ along the x and y axes respectively. If we start from a point, such as the one shown in Figure 1.3, then if η is too large (but still within the stable bounds) the updates will keep zig-zagging their way towards the minima.

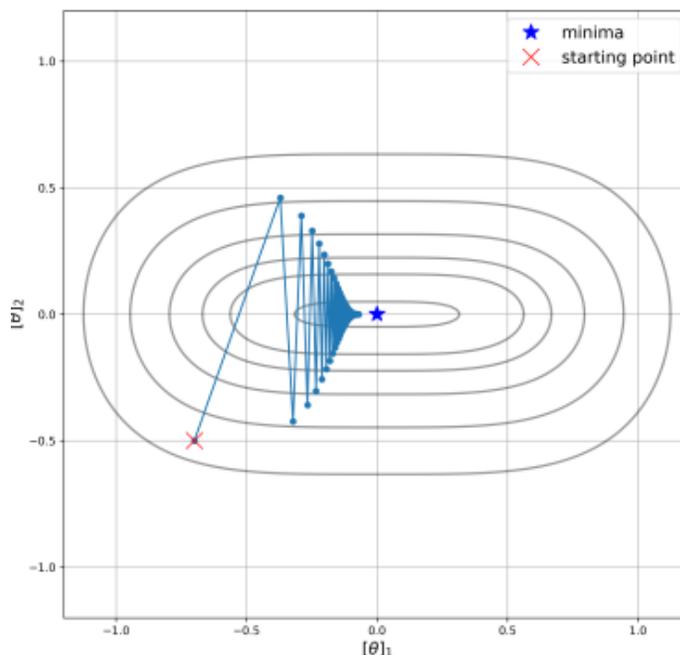


Figure 1.3: GD prefers flatter minima

One of the most popular methods available to handle this problem is Adam optimization.

The main idea behind Adam is to make use of the history of the gradient and the second moment of the gradient.

For an initial learning rate η , the updates are given by

$$\mathbf{g}_k = \beta_1 \mathbf{g}_{k-1} + (1 - \beta_1) \frac{\partial \Pi}{\partial \theta}(\theta_k)$$

$$[\mathbf{G}_k]_i = \beta_2 [\mathbf{G}_{k-1}]_i + (1 - \beta_2) \left(\frac{\partial \Pi}{\partial \theta_i}(\theta_k) \right)^2$$

$$[\boldsymbol{\eta}_k]_i = \frac{\eta}{\sqrt{[\mathbf{G}_k]_i + \varepsilon}}$$

where \mathbf{g}_k and \mathbf{G}_k are the weighted running averages of the gradients and the square of the gradients, respectively. The recommended values for the hyper-parameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 10^{-8}$. Note that the learning rate for each component is different. In particular, the larger the magnitude of the gradient for a component is, the smaller will be its learning rate.

Recalling the example in Figure 1.3, this would mean a lower learning rate for θ_2 compared to θ_1 , and therefore it will help alleviate the zigzag path of the optimization algorithm.

1.6 Calculating gradients using back propagation

The last thing we need to understand is how the gradients are actually evaluated during the training step.

Recall that the output $\mathbf{x}^{(l+1)}$ of layer $l + 1$ is given by

$$\text{Affine transformation: } \xi_i^{(l+1)} = \sum_j W_{ij}^{(l+1)} x_j^{(l)} + b_i^{(l+1)}, \quad 1 \leq i \leq H^{(l+1)} \quad (1.4)$$

$$\text{Non-linear transformation: } x_i^{(l+1)} = \sigma\left(\xi_i^{(l+1)}\right), \quad 1 \leq i \leq H^{(l+1)}. \quad (1.5)$$

Given a training sample (\mathbf{x}, \mathbf{y}) , set $\mathbf{x}(0) = \mathbf{x}$. The value of the loss function (for this particular sample) can be evaluated using the following steps:

1. For $l = 1, \dots, L + 1$
 - (a) Evaluate $\xi^{(l)}$ using (1.4).
 - (b) Evaluate $\mathbf{x}^{(l)}$ using (1.5).
2. Evaluate the loss function for the given sample

$$\Pi(\boldsymbol{\theta}) = \|\mathbf{y} - \mathcal{F}(\mathbf{x}; \boldsymbol{\theta}, \boldsymbol{\Theta})\|^2.$$

This operation can be represented as a computational graph as shown in Figure 1.4. In this diagram, the lower part of the graph corresponds to the evaluation of the loss function Π .

While this process must be repeated for all samples in the training set, the discussion here focuses only on evaluating the loss function and its gradient for a single sample.

In order to update the network parameters, we require $\frac{\partial \Pi}{\partial \boldsymbol{\theta}}$, or more precisely $\frac{\partial \Pi}{\partial \mathbf{W}^{(l)}}$ and $\frac{\partial \Pi}{\partial \mathbf{b}^{(l)}}$ for $1 \leq l \leq L + 1$. We will derive expressions for these derivatives by first deriving expressions for $\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}}$ and $\frac{\partial \Pi}{\partial \mathbf{x}^{(l)}}$.

From the computational graph, it is easy to see how each hidden variable in the network is transformed to the next. Recognizing this, and applying the chain rule repeatedly we arrive at the following expression:

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \frac{\partial \Pi}{\partial \mathbf{x}^{(L+1)}} \cdot \frac{\partial \mathbf{x}^{(L+1)}}{\partial \boldsymbol{\xi}^{(L+1)}} \cdot \frac{\partial \boldsymbol{\xi}^{(L+1)}}{\partial \mathbf{x}^{(L)}} \cdots \frac{\partial \mathbf{x}^{(l+1)}}{\partial \boldsymbol{\xi}^{(l+1)}} \cdot \frac{\partial \boldsymbol{\xi}^{(l+1)}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial \mathbf{x}^{(l)}}{\partial \boldsymbol{\xi}^{(l)}}. \quad (1.6)$$

To evaluate this expression we need to evaluate the following terms:

$$\frac{\partial \Pi}{\partial \mathbf{x}^{(L+1)}} = -2(\mathbf{y} - \mathbf{x}^{(L+1)})^T \quad (1.7)$$

$$\frac{\partial \boldsymbol{\xi}^{(l+1)}}{\partial \mathbf{x}^{(l)}} = \mathbf{W}^{(l+1)} \quad (1.8)$$

$$\frac{\partial \mathbf{x}^{(l)}}{\partial \boldsymbol{\xi}^{(l)}} = \mathbf{S}^{(l)} := \text{diag}[\sigma'(\xi_1^{(l)}), \dots, \sigma'(\xi_{H_l}^{(l)})], \quad (1.9)$$

where the last two relations hold for any network layer l , H_l is the width of that particular layer, and σ' denotes the derivative of the activation function with respect to its argument. Using these relations in (1.6), we arrive at:

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \frac{\partial \Pi}{\partial \mathbf{x}^{(L+1)}} \cdot \mathbf{S}^{(L+1)} \cdot \mathbf{W}^{(L+1)} \cdots \mathbf{S}^{(l+1)} \cdot \mathbf{W}^{(l+1)} \cdot \mathbf{S}^{(l)}. \quad (1.10)$$

Taking the transpose, and recognizing that $\mathbf{S}^{(l)}$ is diagonal and therefore symmetric, we finally get:

$$\frac{\partial \Pi}{\partial \boldsymbol{\xi}^{(l)}} = \mathbf{S}^{(l)} \cdot \mathbf{W}^{(l+1)T} \mathbf{S}^{(l)} \cdots \mathbf{W}^{(L+1)T} \cdot \mathbf{S}^{(L+1)} (-2(\mathbf{y} - \mathbf{x}^{(L+1)})). \quad (1.11)$$

This evaluation can also be represented as a computational graph. In fact, as shown in Figure 1.4, it can be integrated into the original graph,

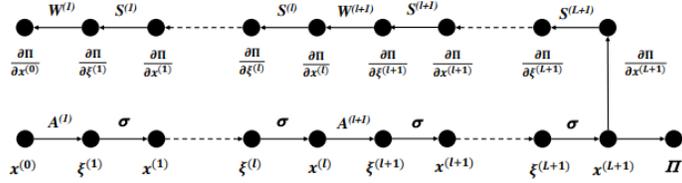


Figure 1.4: Computational graph for computing the loss function and its derivatives with respect to hidden vectors

where this part of the computation appears in the lower branch of the graph. Note that we are now traversing in the backward direction. Hence, the name *back propagation*.

The final step is to evaluate an explicit expression for $\frac{\partial \Pi}{\partial \mathbf{W}^{(l)}}$. This can be done by noting that

$$\frac{\partial \Pi}{\partial \mathbf{W}^{(l)}} = \frac{\partial \Pi}{\partial \xi^{(l)}} \frac{\partial \xi^{(l)}}{\partial W^{(l)}} = \frac{\partial \Pi}{\partial \xi^{(l)}} \otimes \mathbf{x}^{(l-1)}. \quad (1.12)$$

where $[\mathbf{x} \otimes \mathbf{y}]_{ij} = x_i y_j$ is the outer product.

Thus in order to get $\frac{\partial \Pi}{\partial \mathbf{W}^{(l)}}$ we need $\mathbf{x}^{(l-1)}$, which is evaluated during the forward phase and $\frac{\partial \Pi}{\partial \xi^{(l)}}$ which is evaluated during the back propagation.

Chapter 2

Operator Networks

In this chapter, we address the problem of solving a PDE on a given dataset using neural networks.

We begin by introducing a specific class of networks, called *Physics-Informed Neural Networks* (PINNs), providing a brief overview of their structure and functionality. Next, we show how PINNs can be applied to solve PDEs, extend the discussion to the case of parametrized PDEs, and conclude by exploring *Deep Operator Networks* (DeepONet).

2.1 Physics-Informed Neural Networks

The basic idea of PINNs is similar to regression, except that the loss function $\Pi(\boldsymbol{\theta})$ contains derivative operators that arise in the PDE being considered. In the following, we outline the main steps for a general system of PDE. Consider the following problem:

Find $\mathbf{u} : \mathbb{R}^d \rightarrow \mathbb{R}^D$ such that

$$\begin{aligned} L(\mathbf{u}(\mathbf{x})) &= \mathbf{f}(\mathbf{x}), & \mathbf{x} \in \Omega \\ B(\mathbf{u}(\mathbf{x})) &= \mathbf{g}(\mathbf{x}), & \mathbf{x} \in \partial\Omega \end{aligned} \tag{2.1}$$

where L is the differential operator, \mathbf{f} is the known forcing term, B is the boundary operator, and \mathbf{g} is the non-homogeneous part of boundary condition.

To design a PINN for (2.1), the input of the network should be the independent variable \mathbf{x} and the output should be the solution vector \mathbf{u} . Using the network \mathcal{F} , we are looking for a representation of type $\mathbf{u} = \mathcal{F}(\mathbf{x}; \boldsymbol{\theta})$.

The steps would be the following:

1. Construct the loss function:

- Define the interior residual: $R(\mathbf{u}) = L(\mathbf{u}) - \mathbf{f}$.
- Define the boundary residual: $R_b(\mathbf{u}) = B(\mathbf{u}) - \mathbf{g}$.
- Select suitable N_v collocation points in the interior of the domain and N_b points on the domain boundary to evaluate the residuals. These could be chosen on the basis of quadrature rules.

Then the loss function is

$$\begin{aligned}\Pi(\boldsymbol{\theta}) &= \Pi_{\text{int}}(\boldsymbol{\theta}) + \lambda_b \Pi_b(\boldsymbol{\theta}) \\ \Pi_b(\boldsymbol{\theta}) &= \frac{1}{N_b} \sum_{i=1}^{N_b} |R_b(\mathcal{F}(\mathbf{x}_i; \boldsymbol{\theta}))|^2 \\ \Pi_{\text{int}}(\boldsymbol{\theta}) &= \frac{1}{N_v} \sum_{i=1}^{N_v} |R(\mathcal{F}(\mathbf{x}_i; \boldsymbol{\theta}))|^2\end{aligned}$$

2. Train the network: find $\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta})$, and set the solution as $\mathbf{u}^*(\mathbf{x}) = \mathcal{F}(\mathbf{x}; \boldsymbol{\theta}^*)$

We observe that the solution provided by the network is not necessarily the exact solution. In fact $\Pi(\boldsymbol{\theta}^*)$ may not be zero and even if it is, this only implies that the residual vanishes at the collocation point, not everywhere in the domain. Consequently, a good representation of the solution depends, among other factors, on the number of collocation points and how well they cover the domain.

Moreover, note that in the loss function, the term $\Pi_b(\boldsymbol{\theta})$ is crucial for enforcing boundary conditions. Therefore, the hyper-parameter λ_b plays a crucial role in training the network, as it balances the interaction between the two loss terms during the minimization process.

2.1.1 Parametrized PDEs

Assume that the source term in (2.1) is given as a parametric function $\mathbf{f}(\mathbf{x}; \alpha)$. We can train the network to take the parameter into account by considering a network that receives \mathbf{x} and α as input., i.e., $\mathcal{F}(\mathbf{x}, \alpha; \boldsymbol{\theta})$.

Note that we also have to consider the locations for the parameter α while constructing the loss function. If $\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta})$, then the solution to the parameterized PDE would be $\mathbf{u}(\mathbf{x}, \alpha) = \mathcal{F}(\mathbf{x}, \alpha; \boldsymbol{\theta}^*)$. Furthermore, for any new value of $\alpha = \hat{\alpha}$, we could find the solution by evaluating $\mathcal{F}(\mathbf{x}, \hat{\alpha}; \boldsymbol{\theta}^*)$.

The same approach could be used to parameterize the function $\mathbf{g}(\mathbf{x})$, however, a problem soon arises if we are asked to find the solution for an arbitrary non-parametric \mathbf{f} .

A possible way to overcome this issue is to consider an operator that maps a function \mathbf{f} to the solution \mathbf{u} of a given PDE and try to approximate it using a network.

2.2 DeepONet architecture

We describe in the following how to construct a DeepONet, in order to approximate an operator $\mathcal{N} : A \rightarrow U$, where A is a set of functions of the form $a : \Omega_Y \subset \mathbb{R}^d \rightarrow \mathbb{R}$, while U consists of functions of the form $u : \Omega_X \subset \mathbb{R}^D \rightarrow \mathbb{R}$. Furthermore, we assume that point-wise evaluations of both class of functions is possible. The architecture of the DeepONet for this operator is illustrated in Figure 2.1 and it is explained below:

1. Fix M distinct sensor points $\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(M)}$ in Ω_Y .
2. Sample a function $a \in A$ at these sensor points to get the vector

$$\mathbf{a} = [a(\mathbf{y}^{(1)}), \dots, a(\mathbf{y}^{(M)})]^T \in \mathbb{R}^M$$

3. Provide \mathbf{a} as the input to a sub-network, referred to as the *branch net* $\mathcal{B}(\cdot, \boldsymbol{\theta}_B) : \mathbb{R}^M \rightarrow \mathbb{R}^p$, which outputs the vector

$$\boldsymbol{\beta} = [\beta_1(\mathbf{a}), \dots, \beta_p(\mathbf{a})]^T \in \mathbb{R}^p.$$

Here, $\boldsymbol{\theta}_B$ represents the trainable parameters of the branch net.

4. Supply \mathbf{x} as an input to a second sub-network, called the *trunk net* $\mathcal{T}(\cdot, \boldsymbol{\theta}_T) : \mathbb{R}^D \rightarrow \mathbb{R}^p$, whose output would be the vector

$$\boldsymbol{\tau} = [\tau_1(\mathbf{x}), \dots, \tau_p(\mathbf{x})]^T \in \mathbb{R}^p.$$

Here, $\boldsymbol{\theta}_T$ are the trainable parameters of the trunk net.

5. Take a dot product of the outputs of the branch and trunk nets to get the final output of the DeepONet $\tilde{\mathcal{N}}(\cdot, \cdot; \boldsymbol{\theta}) : \mathbb{R}^D \times \mathbb{R}^M \rightarrow \mathbb{R}$ which will approximate the value of $u(\boldsymbol{x})$:

$$u(\boldsymbol{x}) \approx \tilde{\mathcal{N}}(\boldsymbol{x}, \boldsymbol{a}; \boldsymbol{\theta}) = \sum_{k=1}^p \beta_k(\boldsymbol{a}) \tau_k(\boldsymbol{x}) \quad (2.2)$$

where the trainable parameters of the DeepONet will be the combined parameters of the branch and trunk nets, i.e., $\boldsymbol{\theta} = [\boldsymbol{\theta}_T, \boldsymbol{\theta}_M]$.

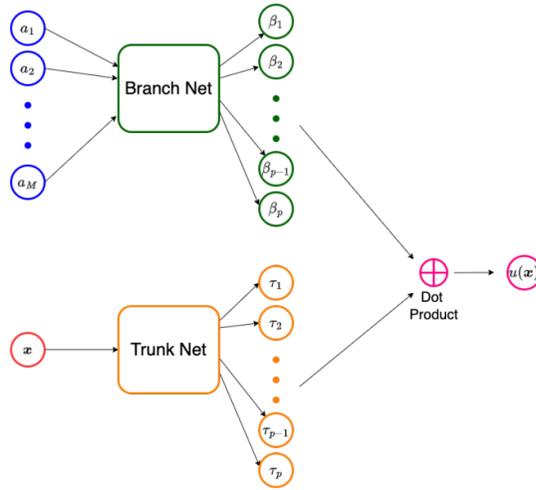


Figure 2.1: Schematic of a DeepONet

In the construction described above, once the DeepONet is trained, it approximates the underlying operator \mathcal{N} , allowing us to estimate the value of $\mathcal{N}(a)(\boldsymbol{x})$ for any $a \in A$ and $\boldsymbol{x} \in \Omega_X$. It is important to note that the M sensor points used in the DeepONet framework must be predefined and remain fixed throughout both the training and evaluation phases.

We now make a few remarks:

- Expression (2.2) has the form of a series of coefficients and functions. In that sense, this formula is similar to the results given by other numerical methods, such as the spectral method and the finite element method. There is, though, a critical difference: while in these methods the basis functions are predetermined, in the DeepONet these functions

are determined by the trunk net and their form depends on the data used to train the network.

- There are essentially two ways of improving the expressivity of a DeepONet: one possibility is to increase the number of network parameters, the other is to increase the dimension p of the vectors formed in these two sub-networks.

2.3 Training DeepONets

Training a DeepONet $\tilde{\mathcal{N}}$, is typically a supervised process that requires pairwise data. The main steps involved are as follows:

1. Select N_1 representative functions $a^{(i)}$, where $1 \leq i \leq N_1$, from the set A . Evaluate these functions at M sensor points, i.e.,

$$a_j^{(i)} = a^{(i)}(\mathbf{y}^{(j)}) \quad \text{for } 1 \leq j \leq M.$$

This gives the vectors:

$$\mathbf{a}^{(i)} = [a^{(i)}(\mathbf{y}^{(1)}), \dots, a^{(i)}(\mathbf{y}^{(M)})]^\top \in \mathbb{R}^M,$$

for each $1 \leq i \leq N_1$.

2. For each $a^{(i)}$, determine (numerically or analytically) the corresponding function $u^{(i)}$ given by the operator \mathcal{N} .
3. Sample the function $u^{(i)}$ at N_2 points in Ω_X , i.e.,

$$u^{(i)}(\mathbf{x}^{(k)}) \quad \text{for } 1 \leq k \leq N_2.$$

Note that one need not choose the same N_2 points across all i in the training set.

4. Construct the training set:

$$S = \{ (\mathbf{a}^{(i)}, \mathbf{x}^{(k)}, u^{(i)}(\mathbf{x}^{(k)})) \mid 1 \leq i \leq N_1, 1 \leq k \leq N_2 \},$$

which consists of $N_1 \times N_2$ samples.

5. Define the loss function as:

$$\Pi(\boldsymbol{\theta}) = \frac{1}{N_1 N_2} \sum_{i=1}^{N_1} \sum_{k=1}^{N_2} \left| \tilde{\mathcal{N}}(\mathbf{x}^{(k)}, \mathbf{a}^{(i)}; \boldsymbol{\theta}) - u^{(i)}(\mathbf{x}^{(k)}) \right|^2.$$

6. Training the DeepONet corresponds to solving the optimization problem:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \Pi(\boldsymbol{\theta}).$$

7. Once trained, for any new $\mathbf{a} \in A$ sampled at the M sensor points (forming the vector $\mathbf{a} \in \mathbb{R}^M$), and for a new point $\mathbf{x} \in \Omega_X$, the corresponding prediction can be evaluated as:

$$u^*(\mathbf{x}) = \tilde{\mathcal{N}}(\mathbf{x}, \mathbf{a}; \boldsymbol{\theta}^*).$$

Remark. The DeepONet can be easily extended to the cases where the input or the output comprises multiple functions.

In the former case the branch network has multiple vectors as input (see Figure 2.2 for the case with two input functions).

In the latter the output of the branch and trunk network leads to D vectors each with dimension p . The solution is then obtained by taking the dot product of each one of these vectors (see Figure 2.3 for the case corresponding to two output functions).

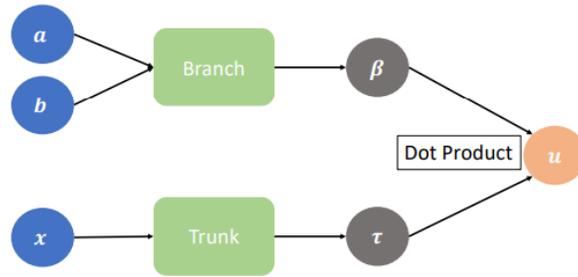


Figure 2.2: Schematic of a DeepONet with two input functions

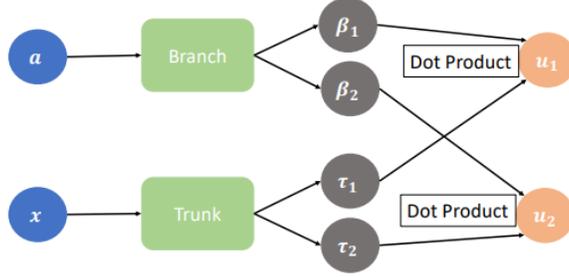


Figure 2.3: Schematic of a DeepONet with two output functions

2.4 Error analysis for DeepONet

There is a universal approximation theorem for a shallow version of DeepONets [1].

Theorem 2.4.1. *Suppose Ω_X and Ω_Y are compact sets in \mathbb{R}^D and \mathbb{R}^d , respectively. Let \mathcal{N} be a nonlinear, continuous operator mapping $V \subset C(\Omega_Y)$ into $C(\Omega_X)$. Then given $\varepsilon > 0$, there exists a DeepONet $\tilde{\mathcal{N}}$ with M sensors and a single hidden layer of width P in the branch and trunk nets such that*

$$\max_{\substack{\mathbf{x} \in \Omega_X \\ a \in A}} |\tilde{\mathcal{N}}(\mathbf{x}, \mathbf{a}; \boldsymbol{\theta}) - \mathcal{N}(a)(\mathbf{x})| < \varepsilon$$

for large enough P and M .

Moreover, the authors in [7] have developed an estimate for the error in a DeepONet in which the different sources of that error are clearly highlighted. The estimate states that:

$$\max_{a \in A} \|\tilde{\mathcal{N}}(\mathbf{x}, \mathbf{a}; \boldsymbol{\theta}) - \mathcal{N}(a)(\mathbf{x})\|_{L^2(\Omega)} \leq C(\varepsilon_h + \sqrt{\varepsilon_t} + \varepsilon_s + M^{-\alpha_1} + N_2^{-\alpha_2}) \quad (2.3)$$

where ε_h represents the numerical solver's error in generating the approximate target solutions $u^{(i)}$ in the training set. The term ε_t denotes the final training error, while ε_s provides an upper bound on the distance between any $a \in A$ and the set of functions $\{a^{(i)}\}_{i=1}^{N_1}$ used to construct the training set, i.e., an estimate of how well the training samples cover the input space A . Additionally, since the input function is evaluated at M discrete sensor nodes, while the output is evaluated at N_2 output nodes, this introduces an extra discretization (or quadrature) error, which is given by the last two terms in (2.3).

Chapter 3

Variationally Mimetic Operator Networks

In the previous chapter, we motivated the study of DeepONet by the fact that we can consider a general PDE, define the operator that maps the parameters of the PDE (functions corresponding to initial conditions, boundary conditions, forcing terms, etc.) into its solution, and approximate that operator with a DeepONet.

In this chapter, we focus on a specific PDE, that is the time-dependent heat equation, and exploit a new operator network, similar to DeepONet, called Variationally Mimetic Operator Network (VarMiON), inspired by the discretization of the variational formulation of the problem.

3.1 PDE model and discretization

Let $\Omega \subset \mathbb{R}^d$ be an open, bounded domain with piecewise smooth boundary. Consider the time-dependent heat conduction equation that describes how heat is transferred within a system over time, in the case with Neumann boundary condition and initial data:

$$\begin{aligned} C(\mathbf{x})\partial_t u(t, \mathbf{x}) - \operatorname{div}(\theta(\mathbf{x})\nabla u(t, \mathbf{x})) &= f(t, \mathbf{x}), \quad \forall (t, \mathbf{x}) \in (0, T) \times \Omega \\ \theta(\mathbf{x})\mathbf{n}(\mathbf{x}) \cdot \nabla u(t, \mathbf{x}) &= g(t, \mathbf{x}), \quad \forall (t, \mathbf{x}) \in (0, T) \times \partial\Omega \\ u(0, \mathbf{x}) &= u_0(\mathbf{x}), \quad \forall \mathbf{x} \in \Omega \end{aligned} \quad (3.1)$$

where:

- $u \in \mathcal{V} \subset L^2((0, T), H^1(\Omega))$ is the temperature field,
- $C \in \mathcal{C} \subset L^2(\Omega) \subset \mathcal{V}$ is the thermal capacity,
- $\theta \in \mathcal{T} \subset L^2(\Omega) \subset \mathcal{V}$ is the thermal conductivity,
- $f \in \mathcal{F} \subset L^2((0, T), H^1(\Omega)) \subset \mathcal{V}$ is the volumetric heat sources,
- $g \in \mathcal{G} \subset L^2((0, T), L^2(\partial\Omega)) \subset \mathcal{V}$ is the flux through the boundary.
- $u_0 \in \mathcal{U} \subset L^2(\Omega) \subset \mathcal{V}$ is the temperature field at time 0

3.1.1 Space-time weak formulation

In this section we want to derive the weak formulation of (3.1).

Let $v \in L^2((0, T), H^1(\Omega))$ be a *test function*, consider the first equation in (3.1) and multiply both sides by v . Then, integrating over Ω and over time, we get

$$\begin{aligned} \int_0^T \int_{\Omega} (C(\mathbf{x})\partial_t u(t, \mathbf{x}) - \operatorname{div}(\theta(\mathbf{x})\nabla u(t, \mathbf{x}))v(t, \mathbf{x}))d\mathbf{x}dt \\ = \int_0^T \int_{\Omega} f(t, \mathbf{x})v(t, \mathbf{x})d\mathbf{x}dt \end{aligned} \quad (3.2)$$

Now, applying divergence theorem to the second term in the LHS of (3.2), we have:

$$\begin{aligned} \int_0^T \int_{\Omega} -\operatorname{div}(\theta(\mathbf{x})\nabla u(t, \mathbf{x}))v(t, \mathbf{x})d\mathbf{x}dt \\ = \int_0^T \left(- \int_{\partial\Omega} \theta(\mathbf{x})\nabla u(t, \mathbf{x}) \cdot \mathbf{n} v(t, \mathbf{x})d\mathbf{x} \right. \\ \left. + \int_{\Omega} \theta(\mathbf{x})\nabla u(t, \mathbf{x})\nabla v(t, \mathbf{x})d\mathbf{x} \right) dt \end{aligned} \quad (3.3)$$

Substituting (3.3) in (3.2) we get

$$\begin{aligned} \int_0^T \int_{\Omega} C(\mathbf{x})\partial_t u(t, \mathbf{x})v(t, \mathbf{x}) + \theta(\mathbf{x})\nabla u(t, \mathbf{x})\nabla v(t, \mathbf{x})d\mathbf{x}dt \\ = \int_0^T \int_{\Omega} f(t, \mathbf{x})v(t, \mathbf{x})d\mathbf{x}dt \\ + \int_0^T \int_{\partial\Omega} \theta(\mathbf{x})\nabla u(t, \mathbf{x}) \cdot \mathbf{n} v(t, \mathbf{x})d\mathbf{x}dt \end{aligned} \quad (3.4)$$

Then we have the weak formulation of (3.1) that is:

Find $u \in L^2((0, T), H^1(\Omega))$ such that

$$\begin{aligned} a(u, v; C, \theta) &= L(v; f, g) & \forall v \in L^2((0, T), H^1(\Omega)) \\ u(0, \mathbf{x}) &= u_0(\mathbf{x}) & \forall \mathbf{x} \in \Omega \end{aligned} \quad (3.5)$$

where

$$\begin{aligned} a(u, v; C, \theta) &= \int_{\Omega} C(\mathbf{x}) \int_0^T \partial_t u(t, \mathbf{x}) v(t, \mathbf{x}) dt d\mathbf{x} + \\ &\quad + \int_{\Omega} \theta(\mathbf{x}) \int_0^T \nabla u(t, \mathbf{x}) \nabla v(t, \mathbf{x}) dt d\mathbf{x} \\ L(v; f, g) &= (v; f) + (v; g)_{\partial\Omega} \\ &= \int_0^T \int_{\Omega} f(t, \mathbf{x}) v(t, \mathbf{x}) d\mathbf{x} dt + \int_0^T \int_{\partial\Omega} g(t, \mathbf{x}) v(t, \mathbf{x}) d\mathbf{x} dt \end{aligned} \quad (3.6)$$

Note that, for the case of heat equation, the variational and weak formulations coincide.

Now taking $\mathcal{X} := \mathcal{F} \times \mathcal{T} \times \mathcal{C} \times \mathcal{G} \times \mathcal{U}$, we can define the *solution operator*:

$$\begin{aligned} \mathcal{S} &: \mathcal{X} \rightarrow \mathcal{V} \\ \mathcal{S}(f, \theta, C, g, u_0) &= u(\cdot; f, \theta, C, g, u_0) \end{aligned} \quad (3.7)$$

which maps the data (f, θ, C, g, u_0) to the unique solution $u(\cdot; f, \theta, C, g, u_0)$ of (3.5).

3.1.2 Discretization of the problem

Our goal is to approximate the operator \mathcal{S} (3.7), using a VarMiON, then we need to consider a discretization of it; we will follow what was done in [9].

First, we define the space \mathcal{V}^h that approximate the space \mathcal{V} in some sense:

$$\mathcal{V}^h = \text{span}\{\phi_l(t, \mathbf{x})\}_{l=1}^L \subset L^2((0, T), H^1(\Omega)) \quad (3.8)$$

In this way any function $v_h \in \mathcal{V}^h$ can be expressed as a linear combination of the finite basis:

$$v_h(t, \mathbf{x}) = \sum_{l=1}^L v_l \phi_l(t, \mathbf{x}) \quad (3.9)$$

We also define the restricted space $\mathcal{V}^h|_{(0,T)\times\partial\Omega} = \{v|_{(0,T)\times\partial\Omega} : v \in \mathcal{V}^h\}$ for the boundary data and the space $\mathcal{V}_0^h = \{v|_{\{t=0\}\times\Omega} : v \in \mathcal{V}^h\}$ to approximate functions that do not depend on time.

We define the space $\mathcal{X}^h := \mathcal{F}^h \times \mathcal{T}^h \times \mathcal{C}^h \times \mathcal{G}^h \times \mathcal{U}^h \subset \mathcal{V}^h \times \mathcal{V}_0^h \times \mathcal{V}_0^h \times \mathcal{V}^h|_{(0,T)\times\partial\Omega} \times \mathcal{V}_0^h$, so that we can consider the projector:

$$\begin{aligned} \mathcal{P} : \mathcal{X} &\rightarrow \mathcal{X}^h \\ \mathcal{P}(f, \theta, C, g, u_0) &= (f^h, \theta^h, C^h, g^h, u_0^h) \end{aligned} \quad (3.10)$$

that approximate the PDE data in \mathcal{X}^h as:

$$\begin{aligned} f^h(t, \mathbf{x}) &= \mathbf{F}^T \Phi(t, \mathbf{x}), \quad C^h(\mathbf{x}) = \mathbf{C}^T \Phi(0, \mathbf{x}), \\ \theta^h(\mathbf{x}) &= \Theta^T \Phi(0, \mathbf{x}), \quad g^h(t, \mathbf{x}) = \mathbf{G}^T \Phi(t, \mathbf{x})|_{(0,T)\times\partial\Omega}, \\ u_0^h(\mathbf{x}) &= \mathbf{U}_0^T \Phi(0, \mathbf{x}) \end{aligned} \quad (3.11)$$

where

- $\Phi(t, \mathbf{x}) = (\phi_1(t, \mathbf{x}), \dots, \phi_L(t, \mathbf{x}))^T$;
- $\mathbf{F} = \mathbf{M}^{-1} \bar{\mathbf{F}}$, $\bar{\mathbf{F}} = \{\bar{F}_i\}$, $\bar{F}_i = (f, \phi_i)$;
- $\mathbf{C} = \{C_i\}$, $C_i = C(\mathbf{x}_i)$;
- $\theta = \{\theta_i\}$, $\theta_i = \theta(\mathbf{x}_i)$;
- $\mathbf{U}_0 = \{U_{0i}\}$, $U_{0i} = u_0(\mathbf{x}_i)$;
- $\mathbf{G} = \tilde{\mathbf{M}}^{-1} \bar{\mathbf{G}}$, $\bar{\mathbf{G}} = \{\bar{G}_i\}$, $\bar{G}_i = (g, \phi_i)_{\partial\Omega}$;
- $\mathbf{M} = \{m_{ij}\}$, $m_{ij} = \int_{\Omega} \int_0^T \phi_i(t, \mathbf{x}) \phi_j(t, \mathbf{x}) dt d\mathbf{x}$;
- $\tilde{\mathbf{M}} = \{\tilde{m}_{ij}\}$, $\tilde{m}_{ij} = \int_{\partial\Omega} \int_0^T \phi_i(t, \mathbf{x}) \phi_j(t, \mathbf{x}) dt d\mathbf{x}$.

Then we can write the discrete version of (3.5) that is:
Find $\mathbf{U} = (U_0, \dots, U_L) \in \mathbb{R}^L$ such that:

$$\begin{aligned} \sum_{l=1}^L a(\phi_j, \phi_l; C^h, \theta^h) U_l &= L(\phi_j; f^h, g^h), \quad \forall j \in \{1, \dots, L\} \\ U_0 &= \mathbf{U}_0 \end{aligned} \quad (3.12)$$

Lastly, system (3.12) can be written in matrix form as:

$$(\mathbf{W}(C^h) + \mathbf{K}(\theta^h))\mathbf{U} = \mathbf{M}\mathbf{F} + \tilde{\mathbf{M}}\mathbf{G} \quad (3.13)$$

with the initial condition $\mathbf{U}^T \Phi(0, \mathbf{x}) = \mathbf{U}_0^T \Phi(0, \mathbf{x})$, where

- $\mathbf{W}(C^h) = \{w_{ij}(C^h)\}$, $w_{ij}(C^h) = \int_{\Omega} C^h(\mathbf{x}) \int_0^T \partial_t \phi_i(t, \mathbf{x}) \phi_j(t, \mathbf{x}) dt d\mathbf{x}$;
- $\mathbf{K}(\theta^h) = \{k_{ij}(\theta^h)\}$, $k_{ij}(\theta^h) = \int_{\Omega} \theta^h(\mathbf{x}) \int_0^T \nabla \phi_i(t, \mathbf{x}) \nabla \phi_j(t, \mathbf{x}) dt d\mathbf{x}$;
- $\mathbf{U} = \mathbf{M}^{-1} \bar{\mathbf{U}}$, $\bar{\mathbf{U}} = \{\bar{U}_i\}$, $\bar{U}_i = (u, \phi_i)$, $i = 0, \dots, L$.

Assuming that the basis $\{\phi_i\}$ is chosen such that $\mathbf{W} + \mathbf{K}$ is invertible for every $(C^h, \theta^h) \in \mathcal{C}^h \times \mathcal{T}^h$, and imposing the initial condition, we finally get the unique solution of (3.12):

$$\mathbf{U} = (\mathbf{W} + \mathbf{K})^{-1}(\mathbf{M}\bar{\mathbf{F}} + \tilde{\mathbf{M}}\bar{\mathbf{G}} - \bar{\mathbf{M}}\mathbf{U}_0) \quad (3.14)$$

where

- $\bar{\mathbf{M}}$ is the matrix with the columns of $\mathbf{W}(C^h) + \mathbf{K}(\theta^h)$ corresponding to the first time instant;
- $\mathbf{W} + \mathbf{K}$ is the matrix with the columns of $\mathbf{W}(C^h) + \mathbf{K}(\theta^h)$ corresponding to the other time instants

From (3.14) we can describe the *numerical solution operator*:

$$\begin{aligned} \mathcal{S}^h : \mathcal{X}^h &\longrightarrow \mathcal{V}^h \\ \mathcal{S}^h(f^h, \theta^h, C^h, g^h, u_0^h) &= u^h(\cdot; f^h, \theta^h, C^h, g^h, u_0^h) \\ &= (\mathbf{W} + \mathbf{K})^{-1}(\mathbf{M}\mathbf{F} + \tilde{\mathbf{M}}\mathbf{G} - \bar{\mathbf{M}}\mathbf{U}_0)^T \Phi(\cdot) \end{aligned} \quad (3.15)$$

that is what we wanted to achieve.

3.2 VarMiON architecture

Motivated by (3.15) we can now describe the architecture of the VarMiON that can be expressed through the following operator:

$$\begin{aligned} \hat{\mathcal{S}} : \mathbb{R}^k \times \mathbb{R}^k \times \mathbb{R}^k \times \mathbb{R}^{k'} \times \mathbb{R}^k &\longrightarrow \mathcal{V}^\tau \\ (\hat{\mathbf{F}}, \hat{\Theta}, \hat{\mathbf{C}}, \hat{\mathbf{G}}, \hat{\mathbf{U}}_0) &\longmapsto (D(L_1 \hat{\mathbf{C}} + L_2 \hat{\Theta})(B_1 \hat{\mathbf{F}} + B_2 \hat{\mathbf{G}} + B_3 \hat{\mathbf{U}}_0))^T \boldsymbol{\tau} \end{aligned} \quad (3.16)$$

where

- \mathcal{V}^τ is the space spanned by the trained trunk functions;
- $\hat{\mathbf{F}} = (f(t_1, \hat{\mathbf{x}}_1), \dots, f(t_M, \hat{\mathbf{x}}_1), \dots, f(t_1, \hat{\mathbf{x}}_k), \dots, f(t_M, \hat{\mathbf{x}}_k))$,
 $\hat{\Theta} = (\theta(\hat{\mathbf{x}}_1), \dots, \theta(\hat{\mathbf{x}}_k))$, $\hat{\mathbf{C}} = (C(\hat{\mathbf{x}}_1), \dots, C(\hat{\mathbf{x}}_k))$,
 $\hat{\mathbf{U}}_0 = (u_0(\hat{\mathbf{x}}_1), \dots, u_0(\hat{\mathbf{x}}_k))$ and
 $\hat{\mathbf{G}} = (g(t_1, \hat{\mathbf{x}}_1^b), \dots, g(t_M, \hat{\mathbf{x}}_1^b), \dots, g(t_1, \hat{\mathbf{x}}_{k'}^b), \dots, g(t_M, \hat{\mathbf{x}}_{k'}^b))$ are given by a *sampling operator*:

$$\begin{aligned} \mathcal{P} : \mathcal{X} &\longrightarrow \mathbb{R}^{M \times k} \times \mathbb{R}^k \times \mathbb{R}^k \times \mathbb{R}^{M \times k'} \times \mathbb{R}^k \\ (f, \theta, C, g, u_0) &\longmapsto (\hat{\mathbf{F}}, \hat{\Theta}, \hat{\mathbf{C}}, \hat{\mathbf{G}}, \hat{\mathbf{U}}_0) \end{aligned} \quad (3.17)$$

and are appropriate sampling of the functions f , θ , C , g , u_0 at the sensor nodes;

- $\mathbf{L}_1, \mathbf{L}_2 \in \mathbb{R}^{k \times k}$, $\mathbf{B}_1 \in \mathbb{R}^{p \times Mk}$, $\mathbf{B}_2 \in \mathbb{R}^{p \times Mk'}$ are linear branches of the network; that is to say, they are learnable matrices;
- $\mathbf{D} : \mathbb{R}^k \rightarrow \mathbb{R}^{p \times p}$ and $\mathbf{B}_3 : \mathbb{R}^k \rightarrow \mathbb{R}^p$ are nonlinear branch;
- $\tau : \mathbb{R}^{d+1} \rightarrow \mathbb{R}^p$ is the trunk subnetwork;
- p is the latent dimension of the network

A visual representation of the VarMiON architecture is shown in Figure 3.1.

To summarize, the process of preparing the data for training the VarMiON is performed through the following steps:

1. For $1 \leq j \leq J$, consider distinct samples $(f_j, \theta_j, C_j, g_j, u_{0j}) \in \mathcal{X}$;
2. The projector (3.10) is used to obtain the discrete approximations $(f_j^h, \theta_j^h, C_j^h, g_j^h, u_{0j}^h) \in \mathcal{X}_h$;
3. The discrete numerical solution $u_j^h = \mathcal{S}^h(f_j^h, \theta_j^h, C_j^h, g_j^h, u_{0j}^h)$ is found;
4. The sampling operator (3.17) is used to generate the VarMiON input vectors $(\hat{\mathbf{F}}, \hat{\Theta}, \hat{\mathbf{C}}, \hat{\mathbf{G}}, \hat{\mathbf{U}}_0)$;
5. A set of output nodes $\{t_k, \mathbf{x}_l\}$, $k = 1, \dots, M$, $l = 1, \dots, L$, is selected, then for each $1 \leq j \leq J$, the numerical solution is sampled at these nodes as $u_{jkl}^h = u_j^h(t_k, \mathbf{x}_l)$;

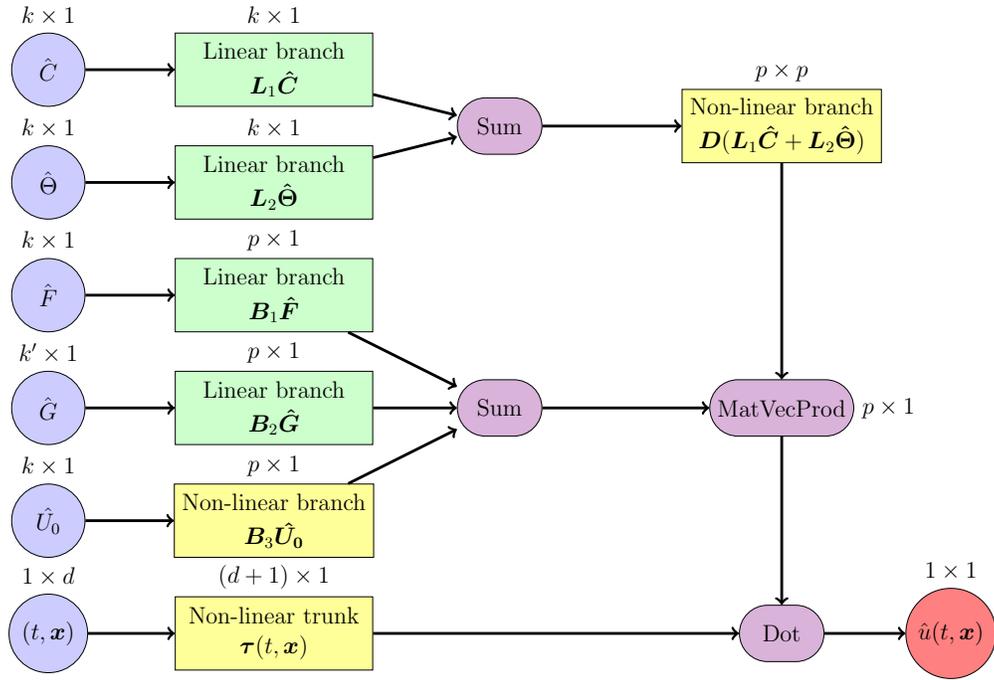


Figure 3.1: VarMiON high-level architecture

6. Finally, all the inputs and output labels are collected in a dataset containing $M \times J \times L$ samples:

$$\mathbb{S} = \left\{ (\hat{\mathbf{F}}_{kl}, \hat{\boldsymbol{\Theta}}_l, \hat{\mathbf{C}}_l, \hat{\mathbf{G}}_{kl}, \hat{\mathbf{U}}_{0l}, t_k, \mathbf{x}_l, u_{jkl}^h) : 1 \leq j \leq J, 1 \leq l \leq L, 1 \leq k \leq M \right\} \quad (3.18)$$

where $(\hat{\mathbf{F}}_{kl}, \hat{\boldsymbol{\Theta}}_l, \hat{\mathbf{C}}_l, \hat{\mathbf{G}}_{kl}, \hat{\mathbf{U}}_{0l})$ is the input to the branch subnets, (t_k, \mathbf{x}_l) is the trunk input, and u_{jkl}^h is the target output.

Now, we look for parameters that minimize the sum of the L_2 norm of the errors $u_j^h - \hat{u}_j$ between the numerical solutions u_j^h and the model predictions $\hat{u}_j = \hat{\mathcal{S}}(\hat{\mathbf{F}}, \hat{\boldsymbol{\Theta}}, \hat{\mathbf{C}}, \hat{\mathbf{G}}, \hat{\mathbf{U}}_0)$ of the instances of (3.18).

If we denote by $\boldsymbol{\psi}$ the set of trainable parameters for the network, which we make explicit by representing the VarMiON by $\hat{\mathcal{S}}_{\boldsymbol{\psi}}$, then we are looking for $\boldsymbol{\psi}^*$ such that:

$$\boldsymbol{\psi}^* = \arg \min_{\boldsymbol{\psi}} \sum_{j=1}^J \int_0^T \int_{\Omega} (u_j^h(\cdot) - \hat{u}_j(\cdot; \boldsymbol{\psi})) \quad (3.19)$$

Then we can define the loss function as:

$$\Pi(\boldsymbol{\psi}) = \frac{1}{J} \sum_{j=1}^J \Pi_j(\boldsymbol{\psi}) \quad (3.20)$$

with $\Pi_j(\boldsymbol{\psi}) = \int_0^T \int_{\Omega} (u_j^h - \hat{\mathcal{S}}_{\boldsymbol{\psi}}(\hat{\mathbf{F}}, \hat{\boldsymbol{\Theta}}, \hat{\mathbf{C}}, \hat{\mathbf{G}}, \hat{\mathbf{U}}_0))$ and obtain the optimization problem:

$$\boldsymbol{\psi}^* = \arg \min_{\boldsymbol{\psi}} \Pi(\boldsymbol{\psi}) \quad (3.21)$$

Chapter 4

Goal oriented VarMiON

In this chapter, we will describe a modification made to the previously discussed VarMiON architecture, to improve the network's accuracy on specific and arbitrary portions of the domain.

This modification specifically concerns the way the loss function is computed.

4.1 Numerical integration

Since the loss function is defined by an integral, we need a cubature rule in order to approximate and compute it.

Let $\Omega = [0, 1]^d$ and $f \in \mathcal{C}((0, T) \times \Omega)$. Our goal is to approximate the integral $\int_0^T \int_{\Omega} f(t, \mathbf{x}) d\mathbf{x} dt$ with the following weighted formula:

$$\int_0^T \int_{\Omega} f(t, \mathbf{x}) d\mathbf{x} dt \approx \sum_{k=1}^M \sum_{l=1}^N w_{kl} f(t_k, \mathbf{x}_l) \quad (4.1)$$

First of all, fix $M \in \mathbb{N} \setminus \{0\}$ and consider a discretization of the time interval $[0, T]$ in M sub-intervals of length $\frac{T}{M}$. Then choose also $n \in \mathbb{N} \setminus \{0\}$ and consider a triangulation of Ω obtained by first taking a uniform grid of $N = (n+1)^d$ points $\mathbf{x}_i \in \Omega$ and then subdividing each d -cube in Ω , with sides of length $h = \frac{1}{n}$, into $d!$ congruent simplices, with a total number of simplices $S = d! \times n^d$. Denote by \mathbf{x}_i^j , $i = 1, \dots, d+1$ the vertices of the simplex s_j for every j . A visual example of this mesh with $n = 9$, is given for the case $d = 3$ in Figure 4.1. Then in Figure 4.2 it is shown how a cube is subdivided

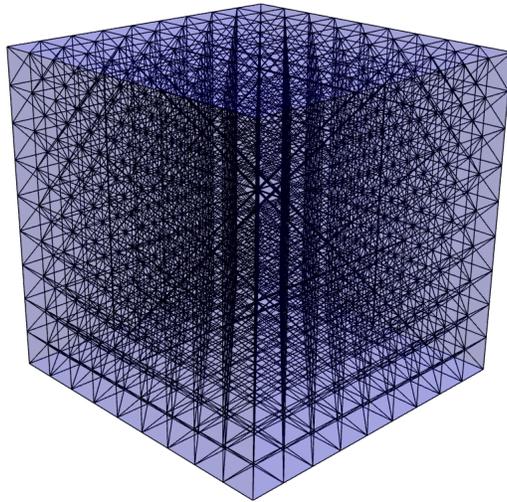


Figure 4.1: Example of uniform mesh with tetrahedral elements on the unit cube. Obtained by first taking a uniform grid with small cubes, then subdividing each cube in six tetrahedra as in Figure 4.2

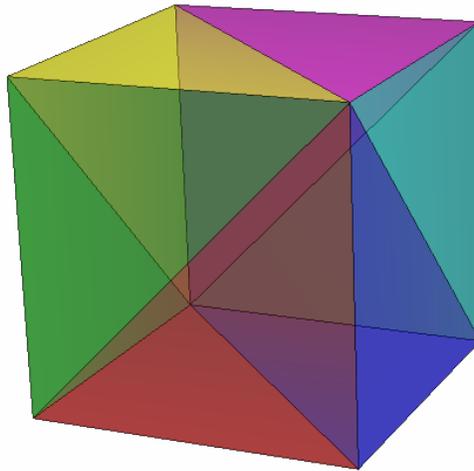


Figure 4.2: Subdivision of a cube in six tetrahedral elements

in tetrahedra. Now we want to approximate the integral above using the trapezoidal rule for both the time integral and the space integral. To do that, we consider the space of polynomial functions in $(0, T) \times \Omega$ of degree at most 1 and take as a basis the set $\{\phi_i^j\}$ with $i = 1, \dots, d+1$ and $j = 1, \dots, S$ where ϕ_i^j is the unique function in that space such that $\phi_i^j(t_k, \mathbf{x}_m^l) = \delta_{im}\delta_{jl}$ for every $m \in \{1, \dots, d+1\}$, $l \in \{1, \dots, S\}$ and $k \in \{1, \dots, M\}$, where δ_{ij} is Kronecker delta.

Then we can write:

$$\begin{aligned}
\int_0^T \int_{\Omega} f(t, \mathbf{x}) d\mathbf{x} dt &\approx \frac{T}{M} \left(\sum_{k=2}^{M-1} \int_{\Omega} f(t_k, \mathbf{x}) d\mathbf{x} + \frac{1}{2} \int_{\Omega} (f(t_1, \mathbf{x}) + f(t_M, \mathbf{x})) d\mathbf{x} \right) \\
&= \frac{T}{M} \sum_{j=1}^S \left(\sum_{k=2}^{M-1} \int_{s_j} f(t_k, \mathbf{x}) d\mathbf{x} + \frac{1}{2} \int_{s_j} (f(t_1, \mathbf{x}) + f(t_M, \mathbf{x})) d\mathbf{x} \right) \\
&\approx \frac{T}{M} \sum_{j=1}^S \sum_{k=2}^{M-1} \left(\int_{s_j} \sum_{i=1}^{d+1} f(t_k, \mathbf{x}_i^j) \phi_i^j(t_k, \mathbf{x}) d\mathbf{x} + \right. \\
&\quad \left. + \frac{1}{2} \int_{s_j} (f(t_1, \mathbf{x}_i^j) \phi_i^j(t_1, \mathbf{x}) + f(t_M, \mathbf{x}_i^j) \phi_i^j(t_M, \mathbf{x})) d\mathbf{x} \right) \\
&= \frac{T}{M} \sum_{j=1}^S \sum_{i=1}^{d+1} \left(\sum_{k=2}^{M-1} f(t_k, \mathbf{x}_i^j) \int_{s_j} \phi_i^j(t_k, \mathbf{x}) d\mathbf{x} + \right. \\
&\quad \left. + \frac{1}{2} (f(t_1, \mathbf{x}_i^j) \int_{s_j} \phi_i^j(t_1, \mathbf{x}) d\mathbf{x} + f(t_M, \mathbf{x}_i^j) \int_{s_j} \phi_i^j(t_M, \mathbf{x}) d\mathbf{x}) \right) \\
&= \frac{T}{M} \sum_{j=1}^S \sum_{i=1}^{d+1} \left(\sum_{k=2}^{M-1} f(t_k, \mathbf{x}_i^j) \frac{\text{area}(s_j)}{d} + \right. \\
&\quad \left. + \frac{1}{2} (f(t_1, \mathbf{x}_i^j) + f(t_M, \mathbf{x}_i^j)) \frac{\text{area}(s_1)}{d} \right) \\
&= \frac{AT}{(d+1)M} \sum_{j=1}^S \sum_{i=1}^{d+1} \left(\sum_{k=2}^{M-1} f(t_k, \mathbf{x}_i^j) + \frac{f(t_1, \mathbf{x}_i^j) + f(t_M, \mathbf{x}_i^j)}{2} \right) \\
&= \frac{AT}{(d+1)M} \sum_{l=1}^{(n+1)^d} \left(\sum_{k=2}^{M-1} f(t_k, \mathbf{x}_l) |N(k, l)| + \frac{f(t_1, \mathbf{x}_l) + f(t_M, \mathbf{x}_l)}{2} |N(1, l)| \right)
\end{aligned}$$

where $A = \frac{1}{S}$ is the area of any s_j , $N(k, l) = \{(t_k, \mathbf{x}_i^j) : (t_k, \mathbf{x}_i^j) = (t_k, \mathbf{x}_l), 1 \leq$

$i \leq d + 1, 1 \leq j \leq S\}$ for every $(k, l) \in \{1, \dots, M\} \times \{1, \dots, N\}$ and $|\cdot|$ denotes the cardinality of a set.

Therefore, we might take as weights

$$w_{kl} = \begin{cases} \frac{T}{S(d+1)M} |N(k, l)|, & k = 2, \dots, M - 1 \\ \frac{T}{2S(d+1)M} |N(k, l)|, & k = 1, M \end{cases} \quad (4.2)$$

4.2 Loss function

The loss function used to train the network is (3.20), here we consider a single instance j of the dataset (3.18):

$$\Pi_j(\boldsymbol{\psi}) = \int_0^T \int_{\Omega} (\hat{u}_j(t, \boldsymbol{x}; \boldsymbol{\psi}) - u_j^h(t, \boldsymbol{x}))^2 d\boldsymbol{x} dt \quad (4.3)$$

where $u_j^h(\cdot)$ is the solution given by FEM method and $\hat{u}_j^h(\cdot; \boldsymbol{\psi})$ is the prediction given by the VarMiON.

Now, consider the quadrature rule (4.1) such that we can approximate the above integral with

$$\int_0^T \int_{\Omega} (\hat{u}(t, \boldsymbol{x}; \boldsymbol{\psi}) - u_j^h(t, \boldsymbol{x}))^2 d\boldsymbol{x} dt \approx \sum_{k=1}^M \sum_{l=1}^N w_{kl} (\hat{u}_j(t_k, \boldsymbol{x}_l; \boldsymbol{\psi}) - u_j^h(t_k, \boldsymbol{x}_l))^2$$

The modification here proposed consists in weighting the prediction error with an appropriate *goal function*, giving to an arbitrary subset of *goal nodes* (i.e. those where greater network accuracy is desired) more importance and grater contribute in the computation of the loss function. An example of a goal function is a function defined on the indices of the mesh nodes, which depends on the distance of each node from the selected subset of the domain.

In particular, we conducted experiments for two cases with the unit cube as a domain, where the network was goal oriented on the facet corresponding to $z = 0$.

In that case, the function used to rescale the loss function is the following:

$$\phi(j) = \alpha^{(1-d(\boldsymbol{x}_j, G))}, \quad j = 1, \dots, (n+1)^3 \quad (4.4)$$

where we denote with $d(\boldsymbol{x}_j, G)$ the distance of the node \boldsymbol{x}_j from the set $G \subset \Omega$, that is where we want the network to be goal oriented. Then,

in order to achieve our goal, we actually modified the quadrature weights, setting $w_{goal_{kl}} = \phi(l)w_{kl}$.

The parameter was experimentally set equal to $\alpha = 10^6$, then all weights were normalized such that the sum was the same as in the case without modification.

Figure 4.3 shows the weights related to the nodes of domain sections perpendicular to the $z - axis$, at a fixed moment in time. The mesh used is the one shown in Figure 4.1.

Only the first 5 sections are shown below, since, after that, the weight distribution is symmetric in the first case and identical to the last in the second case.

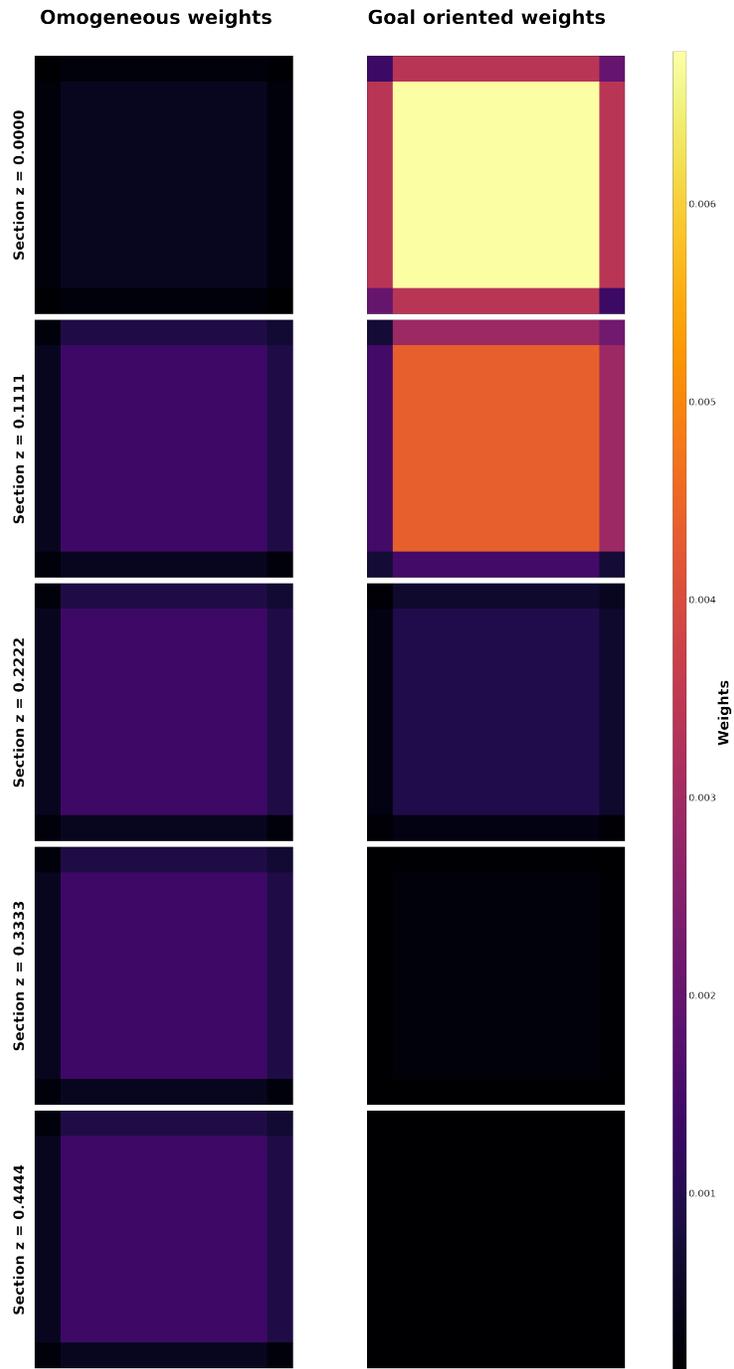


Figure 4.3: Comparison between different weights

4.3 Numerical experiments

In this last section we show results about numerical experiments.

4.3.1 Dataset

We trained a VarMiON to solve the heat equation (3.1) on $[0, T] \times \Omega$, with $\Omega = [0, 1]^3$ and $[0, T] = [0, 2]$, in the case where we want to simulate a thermal flash impacting the surface $z = 0$.

First we solved 80 PDEs over $n_t = 10$ time instants, with parameters defined as follows:

- $u_0(\mathbf{x}) = 0, \quad \forall \mathbf{x} \in \Omega;$
- $C(\mathbf{x}) = 1, \quad \forall \mathbf{x} \in \Omega;$
- $\theta(\mathbf{x}) = 1, \quad \forall \mathbf{x} \in \Omega;$
- $f(t, \mathbf{x}) = 0, \quad \forall (t, \mathbf{x}) \in [0, 2] \times \Omega;$
- $g(t, \mathbf{x}) = \begin{cases} g_0 \exp\left(\frac{\sqrt{(x-x_c(t))^2+(y-y_c(t))^2}}{2\sigma^2}\right) \exp(-\frac{t}{2}) & \text{if } z = 0 \\ 0 & \text{otherwise} \end{cases}$
 $\forall (t, \mathbf{x}) \in [0, 2] \times \partial\Omega, \quad \mathbf{x} = (x, y, z).$

where the following are randomly generated parameters for each PDE:

- $g_0 \in [0.5, 1]$ is the maximum value of the thermal flash;
- $\sigma \in [0.1, 0.3]$ is the decay coefficient of the Gaussian distribution;
- $\mathbf{C}(t) = (x_c(t), y_c(t), 0)$ is the center of the thermal flash.

Each PDE was solved using the FEM method with a tetrahedral mesh on a uniform grid with 12 points per dimension; then, the solutions and the parameters of the PDE were evaluated at sensor nodes chosen as a uniform grid with 10 points per dimension, for a total of $n_p = 10^3 = 1000$ number of evaluation nodes.

Finally, we created the dataset by taking, for each PDE index $i \in \{0, \dots, 79\}$ and each time instant $j \in \{0.2, 0.4, \dots, 2\}$, the parameters of the i -th PDE at time $t = j$ as inputs to the network. We used the solution of this PDE at time $t = j$ as initial data, and set the solution at time $t = j + 1$ as the target output.

4.3.2 Relative error

To evaluate the performance of the models, we first computed the relative L_2 error on the entire domain, then only on the first two sections of the domain. The relative error is defined as:

$$\frac{\left(\int_0^T \int_{\Omega} (u(t, \mathbf{x}; \Theta) - \hat{u}(t, \mathbf{x}))^2 d\mathbf{x} dt\right)^{\frac{1}{2}}}{\left(\int_0^T \int_{\Omega} (\hat{u}(t, \mathbf{x}))^2 d\mathbf{x} dt\right)^{\frac{1}{2}}} \approx \frac{\left(\sum_{k=1}^{n_t} \sum_{l=1}^{n_p} w_{kl} (u(t_k, \mathbf{x}_l; \Theta) - \hat{u}(t_k, \mathbf{x}_l))^2\right)^{\frac{1}{2}}}{\left(\sum_{k=1}^{n_t} \sum_{l=1}^{n_p} w_{kl} (\hat{u}(t_k, \mathbf{x}_l))^2\right)^{\frac{1}{2}}}$$

where the weights used to approximate the two integrals are those given by 4.2.

In table 4.1 is reported the average L_2 relative error of the 80 PDEs used to create the dataset.

VarMiON model	Relative L_2 error measured on Ω	Relative L_2 error measured on D	Relative L_2 error measured on $z = 0$
Uniform version	$1.101 \cdot 10^{-2}$	$2.047 \cdot 10^{-2}$	$2.796 \cdot 10^{-2}$
Goal oriented version	$2.474 \cdot 10^{-2}$	$1.267 \cdot 10^{-2}$	$1.216 \cdot 10^{-2}$

Table 4.1: Comparison of network L_2 relative error with respect to weights used in the loss function, $D = \{x \in \Omega : z = 0 \text{ or } z = 0.1\}$.

From the table, we observe that the modification of the loss function indeed produced a better approximation of the solution near the goal surface, but we lost precision in the rest of the domain.

4.3.3 Analysis of predictions

To visualize the quality of approximation given by VarMiON in its two versions, we consider a particular PDE, defined as in 4.3.1 with $g_0 = 1$, $\sigma = 0.2$, and $\mathbf{C} = (0.5, 0.5, 0)$. Results for this PDE are shown in Figures from 4.4 to 4.6.

What we see in Figures from 4.4 to 4.6 are heat maps representing the temperature field approximated by FEM and VarMiON, with different weights, for the first four sections of the domain, at different points in time.

As expected, we can observe that the predictions given by the VarMiON in the goal oriented version are more accurate than those given by the uniform version in the first two sections of the domain. On the other hand, they become slightly worse as we move away from the surface $z = 0$.

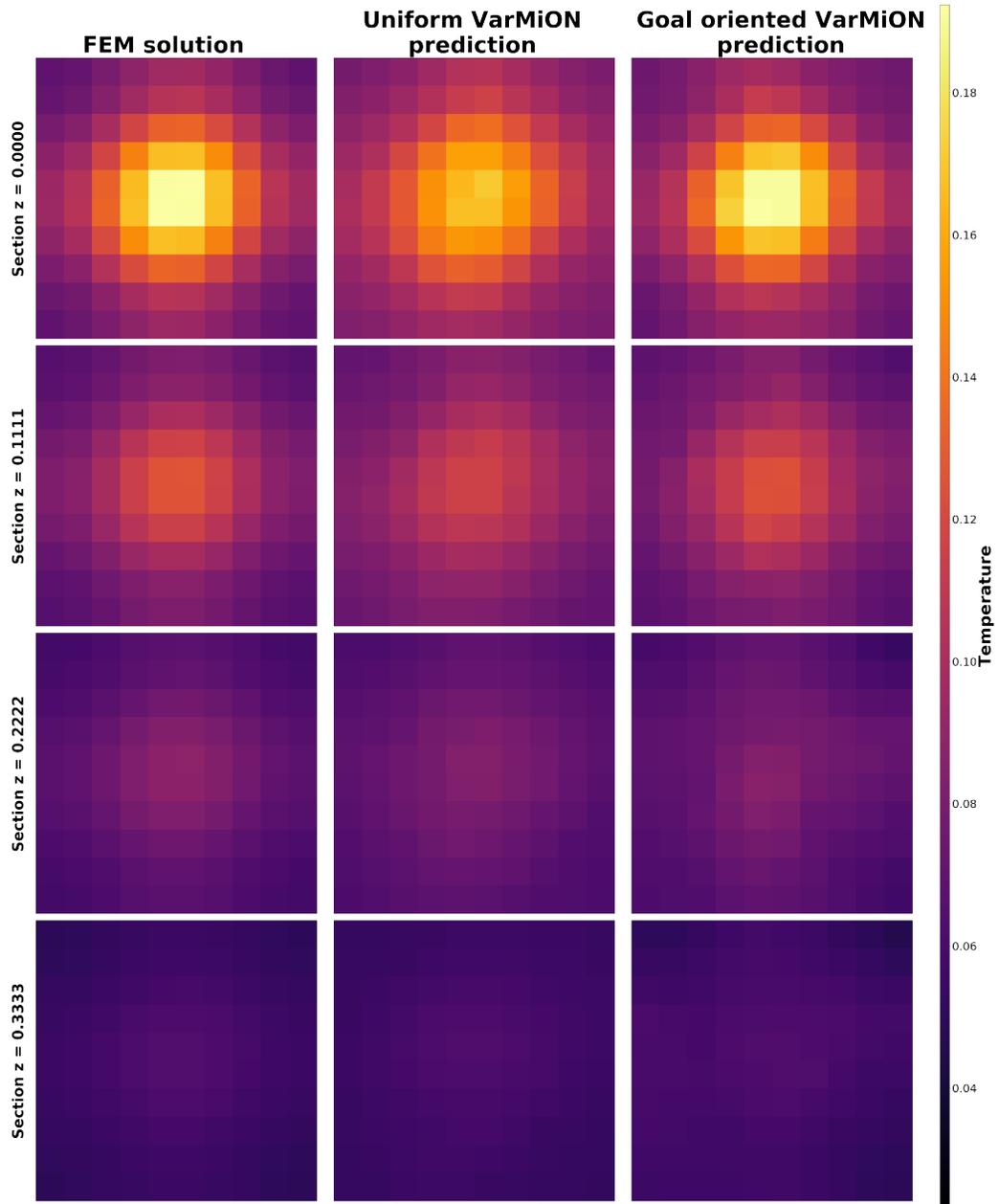


Figure 4.4: Comparison between FEM solution and VarMiON prediction with different weights at time $t=0.2$

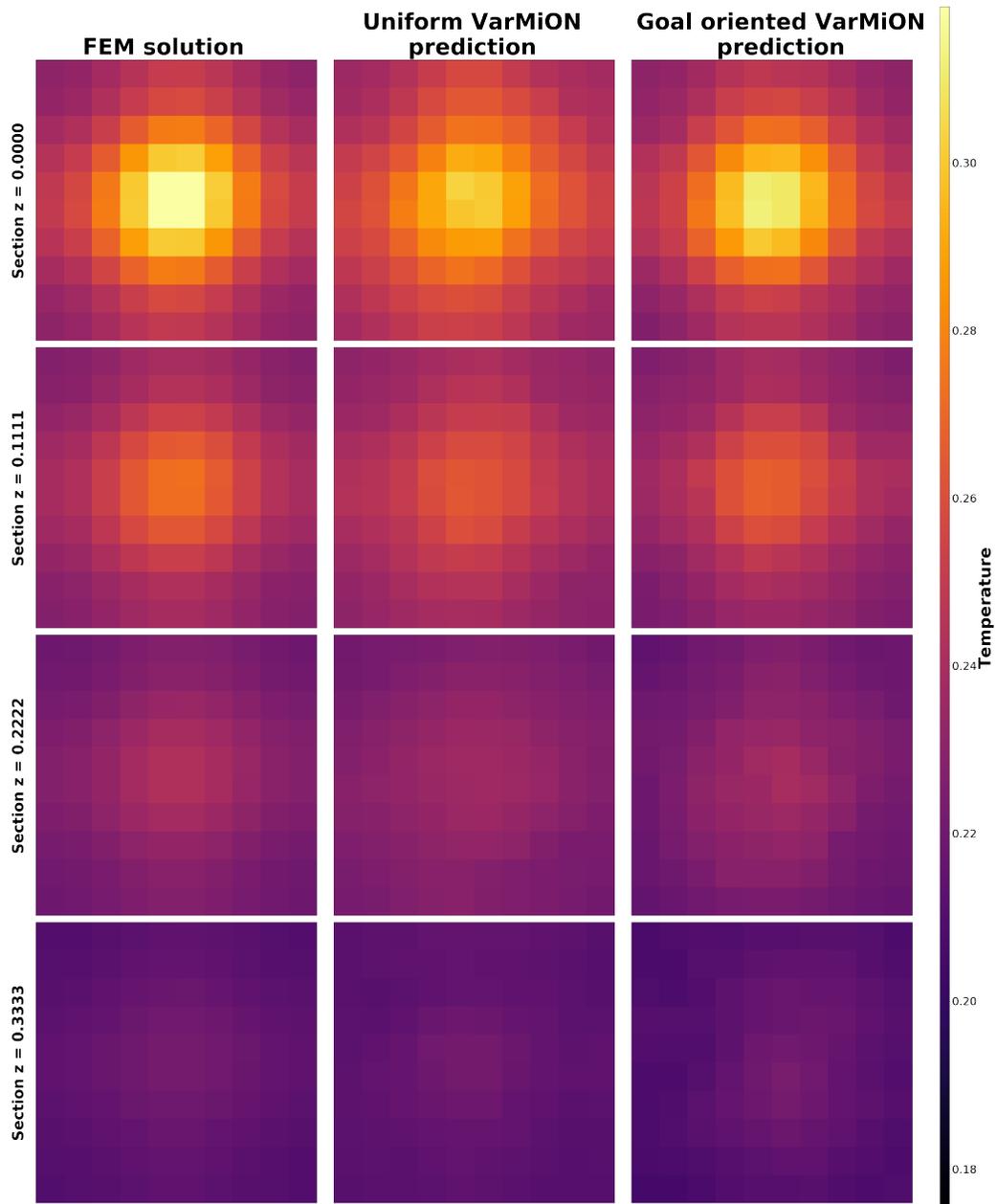


Figure 4.5: Comparison between FEM solution and VarMiON prediction with different weights at time $t=1$

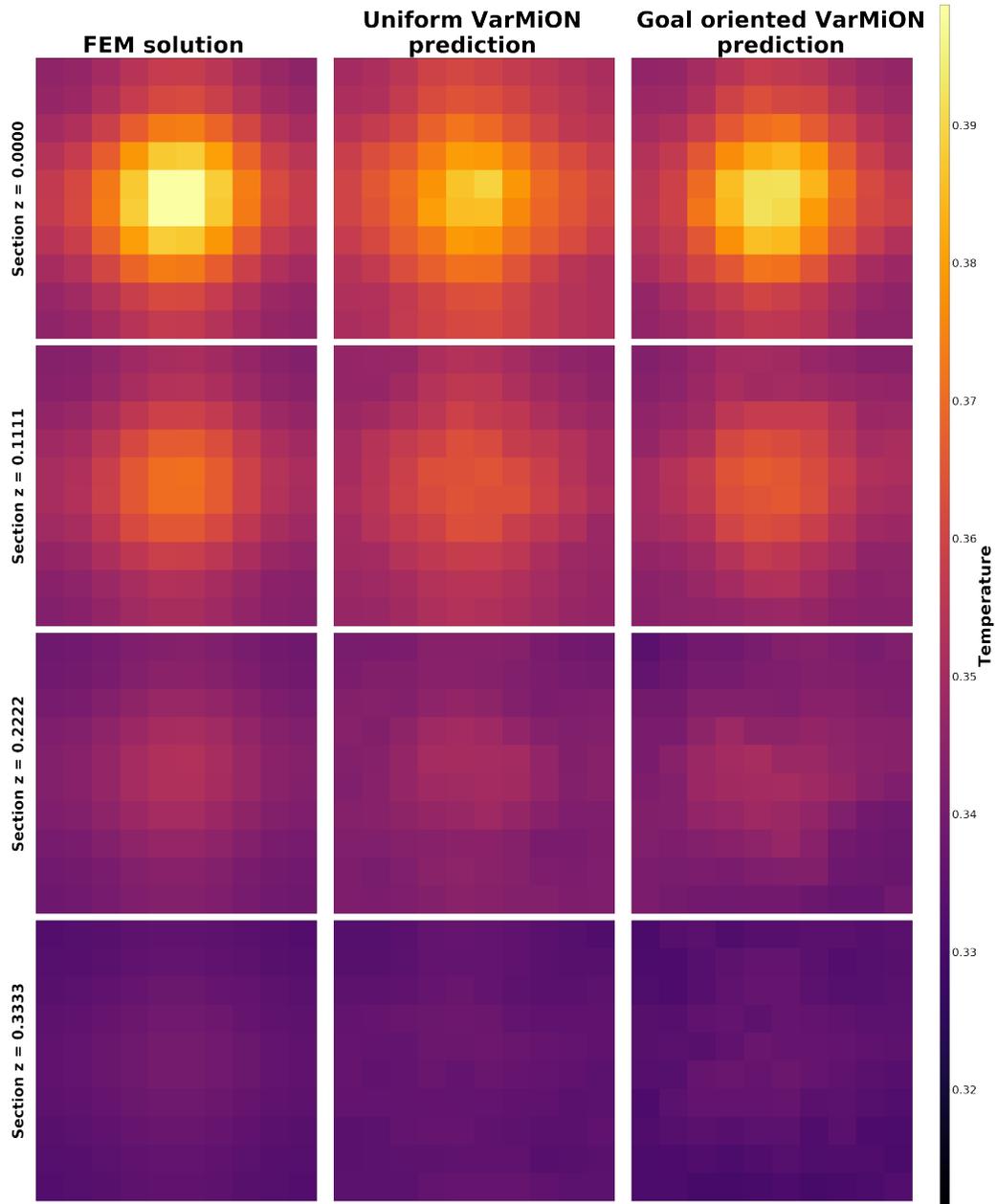


Figure 4.6: Comparison between FEM solution and VarMiON prediction with different weights at time $t=2$

Chapter 5

Kalman filter

In this chapter, we aim to apply the VarMiON to the Kalman Filter (KF) in order to test how the goal oriented modification performs when the network is used to predict the state space in the filter.

5.1 Kalman Filter equations: general case

First of all, we recall the equations of the KF [6].

Consider a Discrete, Linear, Time-Invariant (DLTI) dynamical system in state-space form:

$$\begin{aligned}x(k+1) &= Ax(k) + Bu(k) + v(k) \\ y(k) &= Hx(k) + w(k)\end{aligned}\tag{5.1}$$

where

- $x(k) \in \mathbb{R}^N$ is the *state vector*;
- $u(k) \in \mathbb{R}^{N_u}$ is the *input vector*;
- $y(k) \in \mathbb{R}^{N_y}$ is the *measurement vector*;
- $v(k) \in \mathbb{R}^N$ is the *model error*;
- $w(k) \in \mathbb{R}^{N_y}$ is the *measurement error*;
- A, B are given matrices of adequate dimensions that describe the dynamics of the system;

- H is the *observation matrix* that expresses which states of the system are observable, i.e. it specifies which nodes in the domain receive a direct measurement by sensors.

Then the equations of the KF are given by:

- *Prediction step:*

$$\hat{x}_{k|k-1} = A\hat{x}_{k-1|k-1} + Bu_{k-1} \quad (5.2)$$

$$P_{k|k-1} = AP_{k-1|k-1}A^T + Q \quad (5.3)$$

- *Update step*

$$L_k = P_{k|k-1}H^T(H P_{k|k-1}H^T + R)^{-1} \quad (5.4)$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + L_k(y_k - H\hat{x}_{k|k-1}) \quad (5.5)$$

$$P_{k|k} = P_{k|k-1}(I - K_kH) \quad (5.6)$$

where in the notation $(\cdot)_{k|j}$, k indicates the time instant that we want to estimate, while j indicates up to which time point the information is available to make the estimate. Moreover, we have that:

- $\hat{x}_{k|k-1}$ is the *a priori state estimate*;
- $P_{k|k-1}$ is the *a priori estimate covariance*;
- Q and R are the covariance matrices of v and w respectively;
- L_k is the *Kalman gain*;
- $\hat{x}_{k|k}$ is the *updated state estimate*;
- $P_{k|k}$ is the *updated estimated covariance*.

5.2 Kalman Filter equations: heat equation

In this section, we describe the equations of the KF, in the case where we use the VarMiON architecture to predict the state space in (5.2).

To achieve this, we need a discrete version of (3.1), which is obtained by using the implicit Euler method for the time variable and the FEM for the

space variable [2]. We first take a uniform partition of the time interval $(0, T)$ in $M \in \mathbb{N} \setminus \{0\}$ sub-intervals of length $\Delta t = \frac{T}{M}$, with nodes $t_k, k = 1, \dots, M$, then using the implicit Euler method we obtain the following system:

$$\begin{aligned} u^0(\mathbf{x}) &= u_0(\mathbf{x}) & \mathbf{x} \in \Omega \\ C(\mathbf{x}) \frac{u^{k+1}(\mathbf{x}) - u^k(\mathbf{x})}{\Delta t} + \operatorname{div}(\theta(\mathbf{x}) \nabla u^{k+1}(\mathbf{x})) &= f^{k+1}(\mathbf{x}), & \mathbf{x} \in \Omega \quad (5.7) \\ \theta(\mathbf{x}) \mathbf{n}(\mathbf{x}) \cdot \nabla u^{k+1}(\mathbf{x}) &= g^{k+1}(\mathbf{x}), & \mathbf{x} \in \partial\Omega \end{aligned}$$

for $k \in \{0, \dots, M-1\}$, where we denote by $u^n(\mathbf{x}) = u(t_n, \mathbf{x})$, $f^n(\mathbf{x}) = f(t_n, \mathbf{x})$ and $g^n(\mathbf{x}) = g(t_n, \mathbf{x})$.

Then we can write the semi-discrete formulation of (5.7) that is:
Find $u^k \in H^1(\Omega)$ such that

$$a \left(\frac{u^{k+1}(\mathbf{x}) - u^k(\mathbf{x})}{\Delta t}, v; C, \theta \right) = L(v; f^k, g^k) \quad (5.8)$$

for any $v \in H^1(\Omega)$, and $\forall k \in \{0, \dots, M-1\}$.

Where $a(\cdot, \cdot; \cdot, \cdot)$ and $L(\cdot; \cdot, \cdot)$ are respectively the bilinear form and the linear form associated to (5.7) defined as in (3.6).

Next, we perform a discretization of the space, so we define a mesh on Ω , made by a sequence $T = \{K_i\}_{i=1}^S \subset \Omega$ of closed simplices that forms a *triangulation* of Ω , that is:

1. $K_i \neq \emptyset$ for every $i \in \{1, \dots, S\}$;
2. $K_i \cap K_j = \emptyset$ for every $i, j \in \{1, \dots, S\}$ with $i \neq j$;
3. $\bigcup_{i=1}^S K_i = \bar{\Omega}$;
4. for every $i, j \in \{1, \dots, S\}$ with $i \neq j$, the intersection $K_i \cap K_j$ is either empty or an entire face shared by K_i and K_j .

and we denote by \mathbf{x}_l the nodes of the mesh, for $l = 1, \dots, N$.

Lastly, we define

$$\mathcal{V}^h = \operatorname{span}\{\phi_l(\mathbf{x})\} \subset H^1(\Omega), \quad l = 1, \dots, N \quad (5.9)$$

where ϕ_i is a polynomial of degree at most 1 such that $\phi_i(\mathbf{x}_j) = \delta_{ij}$ for every $i, j \in \{1, \dots, N\}$ and where δ_{ij} is the Kronecker delta.

In this way any function $v_h \in \mathcal{V}^h$ can be expressed as a linear combination of the finite basis:

$$v_h(\mathbf{x}) = \sum_{l=1}^N v_l \phi_l(\mathbf{x}) \quad (5.10)$$

We can now consider the projector:

$$\begin{aligned} \mathcal{P} : H^1(\Omega) &\longrightarrow \mathcal{V}^h \\ v &\longmapsto v_h \end{aligned} \quad (5.11)$$

that approximates a function in $v \in H^1(\Omega)$ with a function $v_h \in \mathcal{V}^h$ in the following way:

$$v(\mathbf{x}) \approx v_h(\mathbf{x}) := \mathcal{P}(v) = \sum_{l=1}^N v(\mathbf{x}_l) \phi_l(\mathbf{x}) \quad (5.12)$$

Using (5.12) in (5.8) we finally get the discrete version of (3.5), which we call *discrete weak formulation*:

Find $v_h^k \in \mathcal{V}^h$ such that

$$a \left(\frac{u_h^{k+1}(\mathbf{x}) - u_h^k(\mathbf{x})}{\Delta t}, v_h^k; C_h^k, \theta_h^k \right) = L(v_h^k; f_h^k, g_h^k) \quad (5.13)$$

for any $v_h^k \in \mathcal{V}^h$ and $\forall k \in \{0, \dots, M-1\}$.

We note that, for any fixed $k \in \{0, \dots, M-1\}$, (5.13) is equivalent to a $N \times N$ linear system, which can be written in matrix form as:

$$(\mathbf{W} + \Delta t \mathbf{K}) \mathbf{U}^{k+1} = \mathbf{M} \mathbf{U}^k + \Delta t (\mathbf{M} \mathbf{F}^k + \mathbf{G}^k) \quad (5.14)$$

where

- $\mathbf{W} = \{w_{ij}\}$, $w_{ij} = \int_{\Omega} C(\mathbf{x}) \phi_i(\mathbf{x}) \phi_j(\mathbf{x}) d\mathbf{x}$;
- $\mathbf{U}^k = \{u_j^k\}$, $u_j^k = u(t_k, \mathbf{x}_j)$;
- $\mathbf{K} = \{k_{ij}\}$, $k_{ij} = \int_{\Omega} \theta(\mathbf{x}) \nabla \phi_i(\mathbf{x}) \nabla \phi_j(\mathbf{x}) d\mathbf{x}$;
- $\mathbf{M} = \{m_{ij}\}$, $m_{ij} = \int_{\Omega} \phi_i(\mathbf{x}) \phi_j(\mathbf{x}) d\mathbf{x}$
- $\mathbf{F}^k = \{f_j^k\}$, $f_j^k = f(t_k, \mathbf{x}_j)$;

- $\mathbf{G}^k = \{g_j^k\}$, $g_j^k = \int_{\partial\Omega} g(t_k, \mathbf{x}) \phi_j(\mathbf{x}) d\mathbf{x}$

for $k = 0, \dots, M$.

Now, we set:

- $x(k) := \mathbf{U}^k$;
- $A := (\mathbf{W} + \Delta t \mathbf{K})^{-1} \mathbf{M}$;
- $B := \Delta t \begin{bmatrix} (\mathbf{W} + \Delta t \mathbf{K})^{-1} \mathbf{M} & (\mathbf{W} + \Delta t \mathbf{K})^{-1} \end{bmatrix}$
- $u(k) := \begin{bmatrix} \mathbf{F}^k \\ \mathbf{G}^k \end{bmatrix}$

for $k = 0, \dots, M$ and we add the error term $v(k)$, which is supposed to be uncorrelated Gaussian noise. Then, we see that (5.14) describes a DLTI system of the form:

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) + v(k) \\ y(k) &= Hx(k) \end{aligned}$$

where the state space $x(k)$ represents the temperature field at time t_k and the input vector $u(k)$ consists of $f(k)$ and $g(k)$, which are respectively the volumetric heat sources at time t_k and the flux through the boundary at time t_k . Moreover, we define the observation matrix H as a matrix with the rows of the identity matrix corresponding to the measured nodes.

Then we can write the equations of the Kalman filter substituting the predictor with a VarMiON model:

- *Prediction step:*

$$\hat{x}_{k|k-1} = \mathcal{F}(\hat{x}_{k-1|k-1}, u_{k-1}) \quad (5.15)$$

$$P_{k|k-1} = (AP_{k|k-1}A^T) + Q \quad (5.16)$$

- *Update step*

$$L_k = P_{k|k-1}H^T(HP_{k|k-1}H^T + R)^{-1} \quad (5.17)$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + L_k(y_k - H\hat{x}_{k|k-1}) \quad (5.18)$$

$$P_{k|k} = P_{k|k-1} - L_kHP_{k|k-1} \quad (5.19)$$

where in (5.15), \mathcal{F} is the VarMiON.

5.3 Numerical experiments

We tested the Kalman filter for the same PDE presented in 4.3.3.

We considered the approximation given by the FEM as the true temperature field and added a Gaussian noise with zero-mean and variance σ_R^2 to simulate experimental measurements affected by error. We defined the observation matrix as the rows of the identity matrix corresponding to the boundary nodes, since we assumed that we would only receive measurements from the boundary.

Then, according to Table 4.1, we defined the covariance matrix of the model error Q as the identity matrix multiplied by σ_Q^2 . In particular we set $\sigma_Q = 10^{-4}$ we used FEM as predictor, $\sigma_Q = 10^{-2}$ for the uniform VarMiON and $\sigma_Q = 2 \cdot 10^{-2}$ for the goal oriented VarMiON. The covariance matrix P of the estimate was also initialized as the identity matrix multiplied by σ_P^2 , assuming to know the initial state error.

We considered two situations: in the first, the noise measurement had a standard deviation $\sigma_R = 10^{-1}$, while in the second, the standard deviation of noise measurement was $\sigma_R = 10^{-3}$.

For both cases, we considered two initial conditions: we passed as initial data to the filter the temperature field that is null throughout the entire domain, with the addition of Gaussian noise with zero-mean and variance $\sigma_P^2 = 10^{-2}$ in the first case, variance $\sigma_P^2 = 10^{-6}$ in the second case.

In all cases, we compared the performance of the filter with the goal oriented VarMiON, first using a uniform Q matrix and then a non-uniform one. The latter was constructed by computing the average relative error of the goal oriented VarMiON for each section of the domain and accordingly modifying the corresponding rows of the Q matrix.

The results presented in the following sections are an average of different realizations of the same experiments.

5.3.1 Case: $\sigma_R^2 = 10^{-2}$, $\sigma_P^2 = 10^{-2}$

This first case was designed to test the performance of different predictors in the presence of both a high initial error and significant measurement noise.

What we observe from 5.1 is that the estimate provided by the goal oriented predictor is better than the one by uniform VarMiON in the first three sections. Additionally, modifying the Q matrix slightly improves the esti-

mates in all sections.

Moreover, the prediction given by goal oriented VarMiON is less affected by initial error than the one given by FEM and uniform VarMiON predictors.

The plots of the relative errors are shown in Figures 5.1-5.4.

Section	FEM	Uniform VarMiON	Goal oriented VarMiON Q uniform	Goal oriented VarMiON Q non-uniform
$z = 0$	0.850 ± 0.186	3.344 ± 0.105	1.412 ± 0.072	1.274 ± 0.059
$z = \bar{1}$	0.937 ± 0.177	2.088 ± 0.170	1.223 ± 0.122	1.168 ± 0.053
$z = \bar{2}$	0.963 ± 0.199	1.587 ± 0.222	1.482 ± 0.127	1.495 ± 0.042
$z = \bar{3}$	1.007 ± 0.299	1.333 ± 0.230	1.674 ± 0.119	1.548 ± 0.106
$z = \bar{4}$	1.118 ± 0.405	1.365 ± 0.237	1.990 ± 0.226	1.671 ± 0.160
$z = \bar{5}$	1.281 ± 0.415	1.380 ± 0.254	2.779 ± 0.255	2.629 ± 0.158

Table 5.1: Mean relative error for the first six sections of the domain for the KF with different predictors. All values are multiplied by 10^{-2}

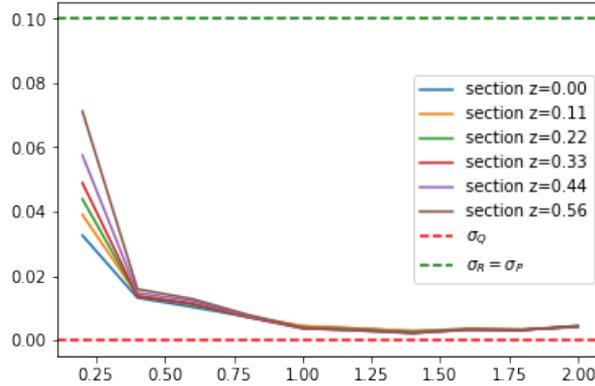


Figure 5.1: Plot of relative error as a function of time, for the first six sections of the domain, for the KF with FEM as predictor

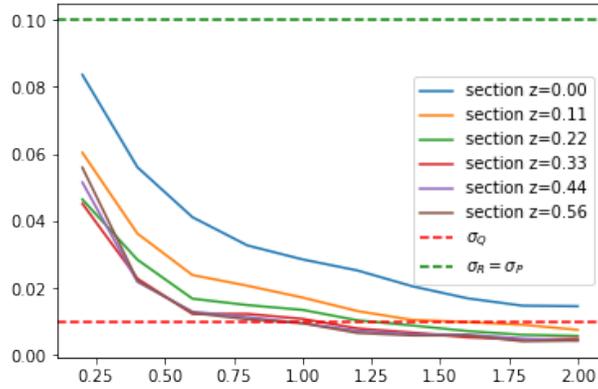


Figure 5.2: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with uniform VarMiON as predictor

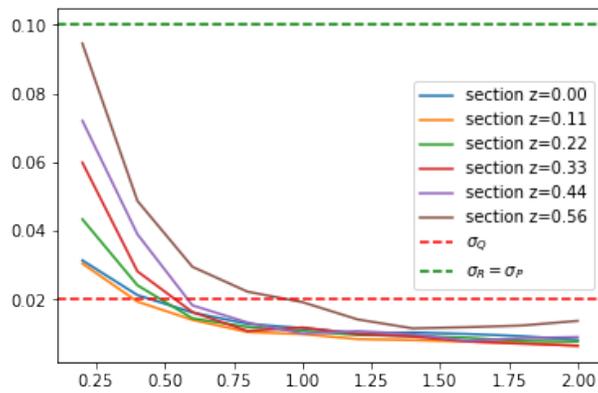


Figure 5.3: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with goal oriented VarMiON as predictor, matrix Q uniform

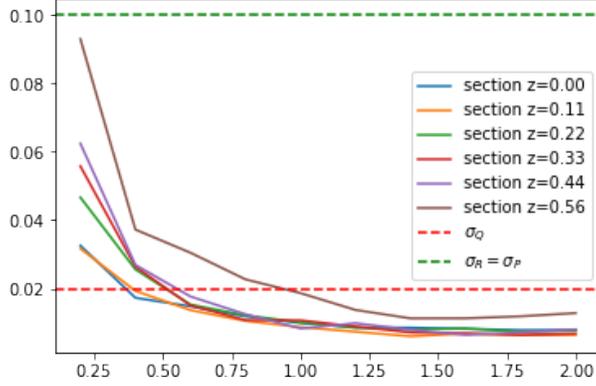


Figure 5.4: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with goal oriented VarMiON as predictor, matrix Q non-uniform

5.3.2 Case: $\sigma_R^2 = 10^{-2}$, $\sigma_P^2 = 10^{-6}$

This second case, instead, aims to test the KF with different predictors in the presence of high measurement noise but a low initial error. This time, what we notice in 5.2 is that the estimate provided by the goal oriented predictor is better than the uniform VarMiON in the first two sections of the domain. Once again, modifying the Q matrix slightly improves the estimates.

The plots of the relative errors are shown in Figures 5.5-5.8.

Section	FEM	Uniform VarMiON	Goal oriented VarMiON Q uniform	Goal oriented VarMiON Q non-uniform
$z = 0$	0.418 ± 0.010	3.263 ± 0.015	1.409 ± 0.031	1.234 ± 0.050
$z = \bar{1}$	0.435 ± 0.008	1.927 ± 0.010	1.415 ± 0.038	1.189 ± 0.086
$z = \bar{2}$	0.328 ± 0.007	1.325 ± 0.010	1.671 ± 0.043	1.509 ± 0.107
$z = \bar{3}$	0.216 ± 0.007	0.942 ± 0.013	1.752 ± 0.061	1.503 ± 0.072
$z = \bar{4}$	0.164 ± 0.013	0.915 ± 0.006	1.754 ± 0.034	1.565 ± 0.033
$z = \bar{5}$	0.155 ± 0.011	0.832 ± 0.029	2.509 ± 0.053	2.471 ± 0.026

Table 5.2: Mean relative error for the first six sections of the domain for the KF with different predictors. All values are multiplied by 10^{-2}

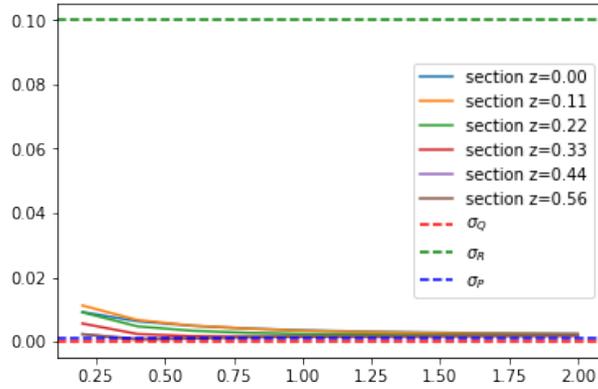


Figure 5.5: Plot of relative error as a function of time, for the first six sections of the domain, for the KF with FEM as predictor

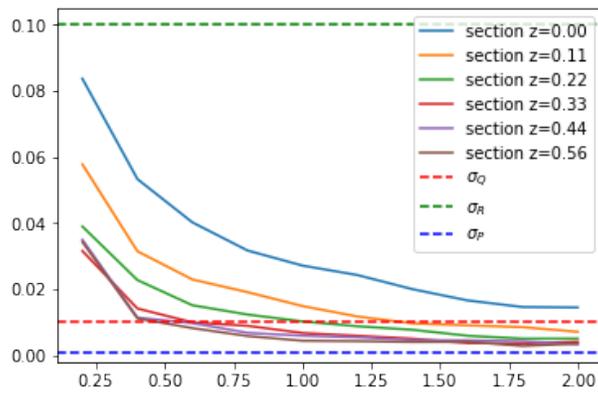


Figure 5.6: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with uniform VarMiON as predictor

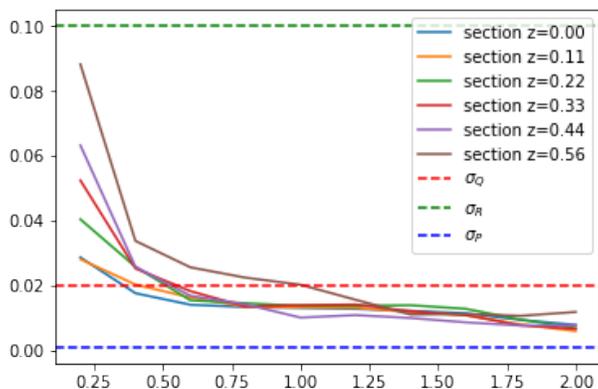


Figure 5.7: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with goal oriented VarMiON as predictor, matrix Q uniform

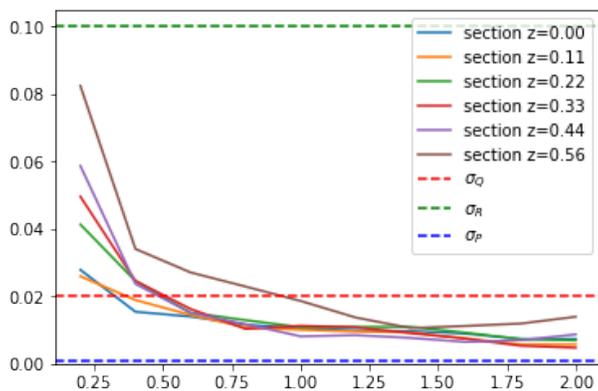


Figure 5.8: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with goal oriented VarMiON as predictor, matrix Q non-uniform

5.3.3 Case: $\sigma_R^2 = 10^{-6}$, $\sigma_P^2 = 10^{-2}$

In this third scenario, the goal is to compare the different predictors in the presence of low measurement noise but a high initial error. What we can observe in 5.3 is that all models perform optimally on the $z = 0$ surface, as accurate measurements are received on it. The goal oriented VarMiON

is better than the uniform version in the second and third sections of the domain, and modifying the Q matrix slightly improves the estimates starting from the fourth section.

The plots of the relative errors are shown in Figures 5.9-5.12.

Section	FEM	Uniform VarMiON	Goal oriented VarMiON Q uniform	Goal oriented VarMiON Q non-uniform
$z = 0$	0.434 ± 0.013	0.425 ± 0.008	0.431 ± 0.005	0.437 ± 0.011
$z = \bar{1}$	0.595 ± 0.043	1.704 ± 0.024	0.785 ± 0.018	0.864 ± 0.005
$z = \bar{2}$	0.613 ± 0.097	1.178 ± 0.023	1.007 ± 0.018	1.043 ± 0.003
$z = \bar{3}$	0.636 ± 0.150	0.877 ± 0.018	1.017 ± 0.006	1.003 ± 0.006
$z = \bar{4}$	0.653 ± 0.190	0.860 ± 0.021	1.254 ± 0.009	1.100 ± 0.002
$z = \bar{5}$	0.658 ± 0.219	0.727 ± 0.020	1.887 ± 0.006	1.744 ± 0.004

Table 5.3: Mean relative error for the first six sections of the domain for the KF with different predictors. All values are multiplied by 10^{-2}

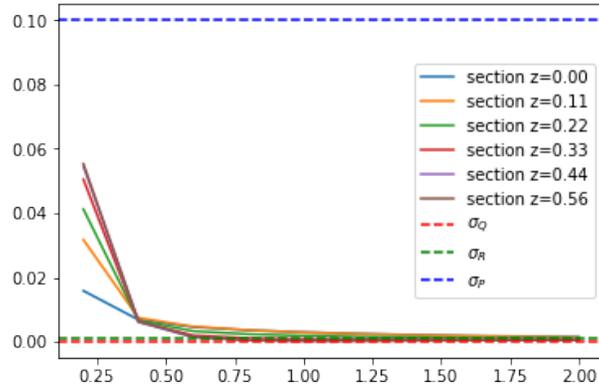


Figure 5.9: Plot of relative error as a function of time, for the first six sections of the domain, for the KF with FEM as predictor

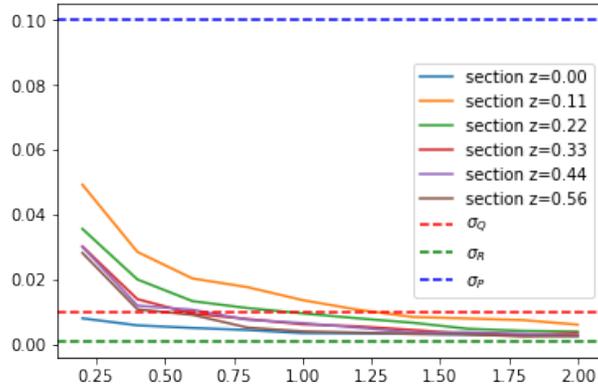


Figure 5.10: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with uniform VarMiON as predictor

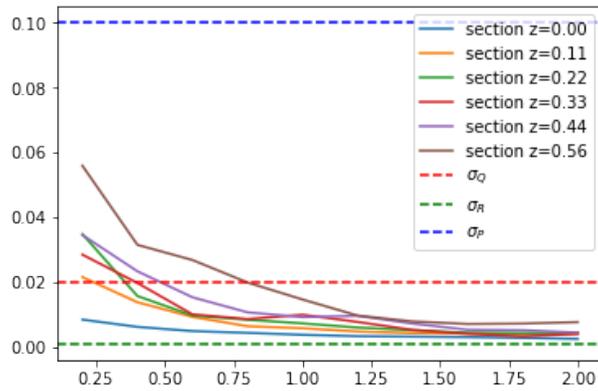


Figure 5.11: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with goal oriented VarMiON as predictor, matrix Q uniform

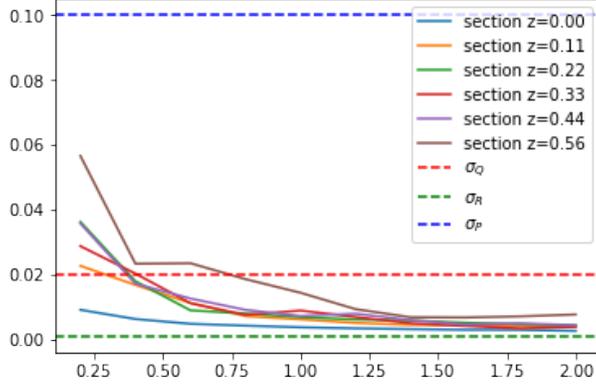


Figure 5.12: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with goal oriented VarMiON as predictor, matrix Q non-uniform

5.3.4 Case: $\sigma_R^2 = 10^{-6}$, $\sigma_P^2 = 10^{-6}$

In this final case, we want to compare the estimate provided by the KF in a scenario where both measurement noise and initial error are low. As in the previous case, we see from 5.4 that all models are equally accurate in the first section of the domain, with the difference that the uncertainty due to the variability of the initial state is no longer present. The goal oriented VarMiON version again proves to be better than the classical version in the second and third sections. As observed in the second case, modifying the Q matrix is not very effective in the presence of low noise, since the variance of this noise remains lower than the model variance.

The plots of the relative errors are shown in Figures 5.13-5.16.

Section	FEM	Uniform VarMiON	Goal oriented VarMiON Q uniform	Goal oriented VarMiON Q non-uniform
$z = 0$	0.369 ± 0.000	0.439 ± 0.015	0.439 ± 0.011	0.432 ± 0.010
$z = \bar{1}$	0.392 ± 0.001	1.654 ± 0.001	0.782 ± 0.006	0.821 ± 0.004
$z = \bar{2}$	0.282 ± 0.001	1.124 ± 0.007	0.975 ± 0.03	0.992 ± 0.002
$z = \bar{3}$	0.160 ± 0.003	0.790 ± 0.006	1.060 ± 0.003	1.045 ± 0.006
$z = \bar{4}$	0.089 ± 0.004	0.785 ± 0.010	1.271 ± 0.007	1.142 ± 0.010
$z = \bar{5}$	0.078 ± 0.005	0.702 ± 0.016	1.889 ± 0.005	1.789 ± 0.011

Table 5.4: Mean relative error for the first six sections of the domain for the KF with different predictors. All values are multiplied by 10^{-2}

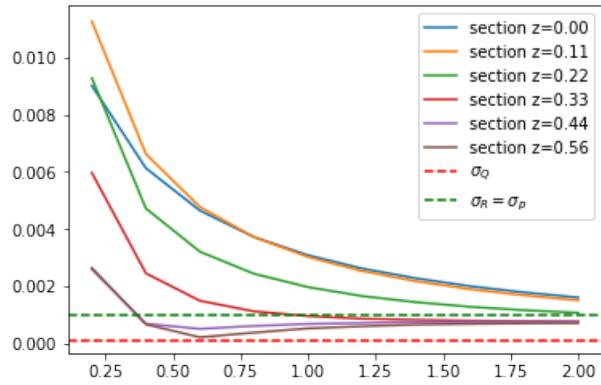


Figure 5.13: Plot of relative error as a function of time, for the first six sections of the domain, for the KF with FEM as predictor

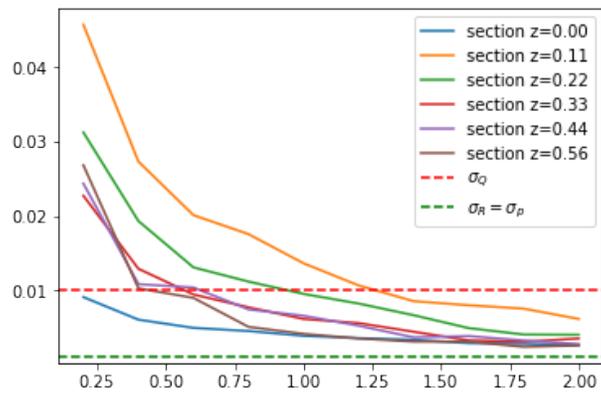


Figure 5.14: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with uniform VarMiON as predictor

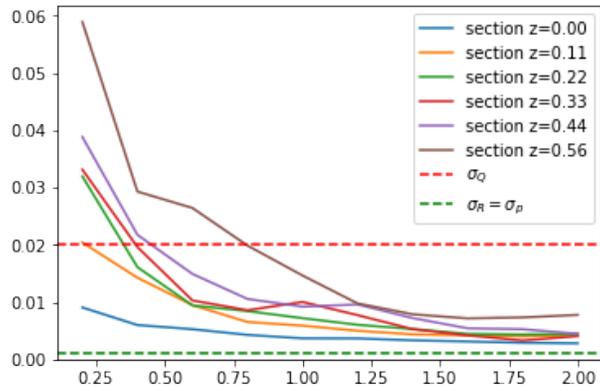


Figure 5.15: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with goal oriented VarMiON as predictor, matrix Q uniform

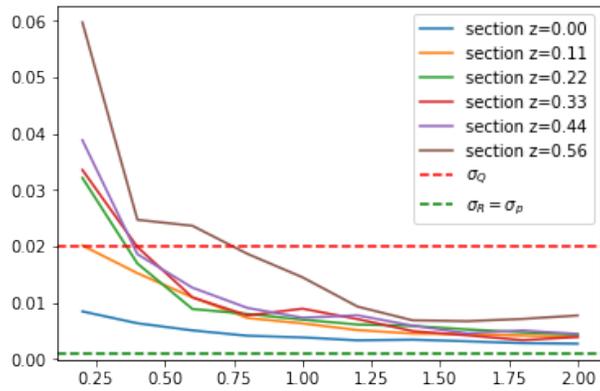


Figure 5.16: Plot of relative error as a function of time, for the first four sections of the domain, for the KF with goal oriented VarMiON as predictor, matrix Q non-uniform

Chapter 6

Conclusions and future developments

In this thesis, we proposed and analyzed a goal oriented version of the VarMiON, an operator network trained for solving PDEs. Experiments made in the case of the heat equation, showed that the goal oriented approach significantly improves the solution accuracy near the reference surface, although at the cost of reduced precision in the rest of the domain.

To mitigate this limitation, we integrated the neural network within a Kalman Filter and evaluated its performance under both high and low noise conditions. The results confirmed that the goal oriented version provides more accurate predictions in the region of interest, even in the presence of noise, while the KF helps correct the estimation in the rest of the domain.

These findings open up several possible directions for future research. One potential improvement could involve modifying the Kalman filter by requiring thermal sensors to collect measurements only in the regions where the model is less accurate, thus enhancing the correction process while reducing the need for extensive measurements.

Another possible development could be the implementation of a deep KF, where the gain matrix becomes a learnable component of the neural network, potentially leading to a more adaptive and efficient filtering approach [3].

Bibliography

- [1] 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, 1995, pp. 911–917, <https://doi.org/10.1109/72.392253>
- [2] M. Dell’Orto, *Variationally Mimetic Operator Neural Networks for the time-dependent heat equation*, Master’s Thesis in Mathematics, Department of Mathematics “Tullio Levi-Civita”– University of Padua, 2024
- [3] E. Chinellato and F. Marcuzzi, *State estimation of partially unknown dynamical systems with a Deep Kalman Filter*, Computational Science, 2024, pp 307-321, https://doi.org/10.1007/978-3-031-63775-9_22
- [4] A. Larese and M. Putti, *Lecture notes of the course ‘Numerical Methods for Differential Equations’*, Department of Mathematics “Tullio Levi-Civita”– University of Padua, 2023
- [5] L. Lu, P. Jin and G. E. Karniadakis, *Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators*, Nat Mach Intell 3, 218-229, 2021, p. 5, <https://doi.org/10.1038/s42256-021-00302-5>
- [6] F. Marcuzzi, *Lecture notes of the course ‘Numerical Linear Algebra and Learning from Data’*, Department of Mathematics “Tullio Levi-Civita”– University of Padua, 2024
- [7] D. Patel, D. Ray, M. R. A. Abdelmalik, T. J. R. Hughes, and A. A. Oberai, *Variationally mimetic operator networks*, Computer Methods in Applied Mechanics and Engineering, 419, 2024 <https://doi.org/10.1016/j.cma.2023.116536>

- [8] D. Ray, O. Pinti and A. A. Oberai, *Deep Learning and Computational Physics*, Springer, 2024, <https://doi.org/10.1007/978-3-031-59345-1>

- [9] L. Rinaldi, *A mathematical construction of the digital twin of bread leavening*, Ph.D. Thesis in Computational Mathematics, Department of Mathematics “Tullio Levi-Civita”– University of Padua, 2024