POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Coupling Ensamble Kalman Filter and Fourier Neural Operators for a comprehensive data assimilation based prediction model

Project for the course Advanced Programming for Scientific Computing

Author: **Minora Adriano - Mondello Malvestiti Giacomo**

Student ID: 10575540
Advisor: Prof. Luca Formaggia
Co-advisors: Stefano Pagani
Academic Year: 2022-2023

Abstract

The integration of Ensemble Kalman Filters (EnKF) with Fourier Neural Operators (FNO) marks a significant advancement, offering robust solutions for complex data assimilation challenges. EnKF, renowned for its effectiveness in sequential data assimilation is explored for its capacity to handle nonlinear dynamics and chaotic systems. On the other hand, FNO emerges as a promising tool in learning continuous mappings of functional spaces, vital for managing high-dimensional datasets. The fusion of these methodologies promises a transformative approach in predictive accuracy, extending its potential applications from weather forecasting to environmental science. The aim of this report is to establish a robust, scalable approach to data assimilation that can adapt to the evolving complexities of real-world datasets, providing a significant step forward in computational science and predictive analytics.

Contents

Introduction

Data assimilation is a predictive modeling technique that provide a crucial bridge between theoretical models and real-world observational data. The integration of Ensemble Kalman Filters (EnKF) with Fourier Neural Operators (FNO) represents a novel approach in this domain. This report presents a comprehensive exploration of this integration.

Ensemble Kalman Filters are known for their ability to provide real-time updates to predictive models. The EnKF technique represents a dynamic procedure capable of assimilating observational data into models, accounting for uncertainties in both the model and observations. Fourier Neural Operators, on the other hand, represent a new class of deep learning models designed to learn mappings in functional spaces. FNO's ability to handle spatial complexity and manage high-dimensional datasets makes it an good candidate for integration with EnKF.

The coupling of EnKF and FNO is proposed as a new framework intended to expand traditional data assimilation methods. This report presents an implementation strategy for such a coupled model along with some numerical examples. It aims toprovide a robust and scalable solution to the complex challenges of data assimilation.

1 | Data Assimilation via EnKF

1.1. Generalities on data Assimilation

Data assimilation is a mathematical time-stepping procedure used for determining the optimal state estimate of a system and that can also well interpolate sparse information data using knowledge of the system being observed. It is indeed very helpful for numerical forecasts. This technique works well even with chaotic dynamical system where small changes in initial values lead to large ones in the predictions and for this reason classical interpolation methods are not that effective.
Data assimilation consists on a step-by-step upgrade of the state estimate, which is the result of a forecast based on previous step conditions. To this estimate is then applied a correction based on observed data and estimated errors that are present in both the observations and the forecast itself. This can be done through a weighting factor that is applied to the difference between the forecast and the observations at that time and it determines how much of a correction should be made to the forecast based on the new information from the observations.

The first step in the mathematical formalisation of the analysis is the definition of the mathematical quantities needed to represent the phisical model. The collection of numbers needed to represent the state of the model is collected as a column matrix called the state vector $\mathbf{x}$, with dimension that will be denoted as N. As long as it is impossible to perfectly replicate reality, the true state at the time of analysis will be $\mathbf{x}_t$, this state vector tries to best replicate the true state at the given time. Another important value of the state vector is $\mathbf{x}_b$, the background estimate of the true state before the analysis is carried out, valid at the same time. The analysis problem is to find a correction $\delta\mathbf{x}$ such that:

$$\mathbf{x}_a = \mathbf{x}_b + \delta\mathbf{x}, \tag{1.1}$$

where $\mathbf{x}_a$, called "analysis", must be as close as possible to $\mathbf{x}_t$ [6].

1.2. Ensemble Kalman Filter models

Ensemble Kalman Filter method is a variant of the more general Kalman filter method: this method provide a trade-off between the expectations of the actual state of a system and measurements [20]. The actual state of the system is obtained from physical or data-provide models and it is assumed that extracting information from the process does not involve perturbations in the model itself. The aim of this methods is then to predict the true state of the phenomenon and to correct the prediction made.

It is a two stage process: the first stage is the prediction, the second one is the correction. Prediction is based on the estimate of the system last state, and can depend also on parameters or control inputs, which are imposed changes to the system by external factors at a certain time. The second phase involves the correction of the prediction on the basis of the measurement data. Effectively this estimate is a weighted average of a priori state estimates and measurements. It is therefore more representative of the actual state of the system taking into account possible model discrepancy with respect to the real phenomenon.

For evolutive systems the error in the predicted state estimate, which is the difference between predicted and actual state, is expected to increase in time due to the propagation of errors. For this reason it is crucial that predicted state estimate is properly corrected and can then be used as base for new predictions: correction grants the reliability of this method. Both two process are characterized by the presence of a noise which is taken into account by appropriate covariance matrices.

Different kind of Kalman Filters have been developed, in this work the analysis will be focused on the so called Ensemble Kalman Filter (EnKF), but first there will be presented the main feature of the Basic Kalman Filter and at last it will be explained why the EnKF represented the best option with respect to the other filters.

1.2.1. Basic Kalman Filter

The aim of the first phase of this method is the prediction of the state vector $\hat{\mathbf{x}}$ and of the covariance matrix $\mathbf{P}$. These predictions are based on the state vector and covariance matrix at time $k-1$ that are respectively $\hat{\mathbf{x}}_{k-1}$ and $\mathbf{P}_{k-1}$, and are carried on through the state transition matrix $\mathbf{F}$, a $N \times N$ matrix, which governs the theoretical evolution of the state vector from $\hat{\mathbf{x}}_{k-1}$ to $\hat{\mathbf{x}}_k$. Sometimes, in the prediction phase, there is another matrix which can be involved: the control input matrix $\mathbf{B}_k$. This is a $N \times L$ matrix where L is the dimension of the control input vector. It takes into account the external actions which

take part in the evolutive process (i.e. a sudden introduction of preys in the prey-predator process).
Once the prediction is made, and recall that this is strictly dependent on the initial state of the vector, it has to be corrected (correction phase). This is made by the state-to-measurement transformation matrix $\mathbf{H}$. It is then reached the corrected prediction $\hat{\mathbf{x}}_k$. The idea of this phase is then to blend the apriori estimate represented by the first prediction and the measurement at the current time instant: this minimize the difference between the found aposteriori estimate and the actual state vector. The covariance matrices, $\mathbf{Q}_k$ for the process and $\mathbf{R}_k$ for the measurements, take into consideration the uncertainties related to the predictions and the measurements.

For Basic Kalman Filter, the relation that links the actual state vector $\hat{\mathbf{x}}_k$ with the one at the precedent time $\hat{\mathbf{x}}_{k-1}$ is the following:

$$\mathbf{x}_k = \mathbf{F}_k\mathbf{x}_{k-1} + \mathbf{B}_k\mathbf{u}_k + \mathbf{q}_k, \tag{1.2}$$

where $\mathbf{q}_k$ is the process noise which is the result of a zero mean multi variate normal distribution with covariance matrix $\mathbf{Q}_k$. On the other hand, the relation between the actual state vector $\hat{\mathbf{x}}_k$ and the measurement done at time k is:

$$\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{r}_k, \tag{1.3}$$

where $\mathbf{r}_k$ is the measurement noise which is the result of a zero mean multi variate normal distribution with covariance matrix $\mathbf{R}_k$.

All the steps for prediction and correction phase will be now presented, but before that, consider the difference of notation that is introduced now:

$$\hat{\mathbf{x}}_{k-1|k-1}, \tag{1.4}$$

where the left part of the subscript has the same meaning as before, which indicates at what time our quantity is taken, and the right of the subscript part indicates that the information, on which the prediction is based, is taken at time $k-1$.
For what concerns the aposteriori estimates:

$$\hat{\mathbf{x}}_{k|k}, \tag{1.5}$$

the "input" information is taken at time k, being the prediction estimate, and it is corrected still at time k.

In the prediction phase the two equations regarding apriori state estimates and apriori covariance matrix are respectively:

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}_k \hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}_k \mathbf{u}_k \tag{1.6}$$

and

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_K^T + \mathbf{Q}_k. \tag{1.7}$$

In order to recover now the aposteriori state estimate and the aposteriori covariance matrix, the following quantities are introduced:

$$\hat{\mathbf{z}}_k = \mathbf{H}_k \hat{\mathbf{x}}_{k|k-1}, \tag{1.8}$$

where the state to measurement transformation matrix is applied to the apriori estimate (indeed the aim of this phase is to correct the first estimate). The residuals:

$$\tilde{\mathbf{z}}_k = \mathbf{z}_k - \hat{\mathbf{z}}_k, \tag{1.9}$$

the covariance matrix of the residuals ($K \times K$ matrix):

$$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^T + \mathbf{R}_k, \tag{1.10}$$

the optimal gain ($K \times K$ matrix):

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^T \mathbf{S}_K^{-1}. \tag{1.11}$$

It is now possible to define the aposteriori state estimate:

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \tilde{\mathbf{z}}_k, \tag{1.12}$$

and the aposteriori covariance matrix:

$$\mathbf{P}_{k|k} = (\mathbf{I}_N - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k|k-1}, \tag{1.13}$$

where $\mathbf{I}_N$ is the identity matrix of dimension $N \times N$.
According to [13], relation (1.12) can be rewritten in the following way:

$$\mathbf{x}_{k|k} = (\mathbf{P}_{k|k-1}^{-1} \mathbf{x}_{k|k-1} + \mathbf{H}_k^T \mathbf{R}_k^{-1} \mathbf{z}_k) \tag{1.14}$$

and

$$\mathbf{P}_{k|k}^{-1} = \mathbf{P}_{k|k-1}^{-1} + \mathbf{H}_k^T \mathbf{R}_K^{-1} \mathbf{H}^T. \tag{1.15}$$

Equation (1.15) is obtained from equation (1.13) using the Sherman–Morrison–Woodbury formula. The last two equations give us a nice interpretation of the Kalman filter update. In equation (1.14) it is possible to see that the aposteriori state estimate is taken as a weighted average of the apriori state estimate $\mathbf{x}_{k|k-1}$ and of the measurement vector $\mathbf{z}_k$ where the weights are proportional to the prior covariance matrix $\mathbf{P}_{k|k-1}^{-1}$ and to a a combination of the observation matrix and the data precision matrix $\mathbf{H}_k^T \mathbf{R}_k^{-1}$. The aposteriori covariance in (1.15) is the sum of the apriori covariance and the data precision (projected onto the state space).

Generally Basic Kalman Filter yields to good prediction when the dynamics of the system is known and it is linear, the noise (both of the process and the measurement) is Gaussian (also white for the process) and the covariance matrices are known. The performance of Basic Kalman Filter worsen if the system is highly nonlinear or if the assumptions about the noise are violated. When dealing with this cases other methods different from the basic one have to be used. In this work, the chosen method is the Ensemble Kalman Filter (EnKF).

1.2.2. Ensemble Kalman Filter

The Ensemble Kalman Filter is an extension of the Kalman filter that is specifically designed for nonlinear and non-Gaussian systems. It is particularly useful for problems where the underlying dynamics are complex and may not be fully known. A fundamental distinction between the two methodologies lies in the fact that the Basic Kalman filter explicitly deals with the entire distribution of the state, whereas the Ensemble Kalman Filter works with an ensemble of vectors that approximates the state distribution. This representation of the state as an ensemble amounts to a form of dimension reduction, as it involves propagating only a limited ensemble rather than the complete joint distribution, which would include the entire covariance matrix. A key aspect of the EnKF, which distinguish it from most of other Monte Carlo algorithms, is that it takes advantage of a linear updating rule that converts the prior ensemble to a posterior ensemble after each observation. The use of shifting rather than reweighting during the update step enables the algorithm to maintain stability, especially in high-dimensional problems.

EnKF still has two phases like the basic Kalman filter. They still consist on a prediction step and on a correction one for each time of the simulation. Predictions and corrections are made on N vectors which compose the ensemble. If N tends to infinity EnKF converges

to the exact Kalman Filter for linear Gaussian models but in practice when N grows too much the system becomes unworkable.

The difference of notation with respect to the basic Kalman filter is the introduction of the vectorial notation. Furthermore, the state transition matrix and the state to measurement transformation matrix are now replaced by two functions that respectively propagates each member of the ensemble forward using the nonlinear system model and that correct the prediction member by member. It is needed because, as discussed before, both prediction and correction are made on each sample that compose the ensemble $(\mathbf{x}_k^{(i)}, ..., \mathbf{x}_k^{(N)})$ and equations become:

$$\mathbf{x}_k^{(i)} = \mathbf{f}_k(\mathbf{x}_{k-1}^{(i)}) + \mathbf{b}_k(\mathbf{u}_k^{(i)}) + \mathbf{q}_k^{(i)} \tag{1.16}$$

and

$$\mathbf{z}_k^{(i)} = \mathbf{h}_k(\mathbf{x}_k^{(i)}) + \mathbf{r}_k^{(i)} \tag{1.17}$$

Where the $\mathbf{f}_k$, $\mathbf{h}_k$ and $\mathbf{b}_k$ are respectively the state transition function, the state to measurement transformation function and the control input function, defined as:

$$\mathbf{f}_k(\mathbf{x}_k) = \mathbf{F}_k\mathbf{x}_k \tag{1.18}$$

$$\mathbf{h}_k(\mathbf{x}_k) = \mathbf{H}_k\mathbf{x}_k \tag{1.19}$$

$$\mathbf{b}_k(\mathbf{u}_k) = \mathbf{B}_k\mathbf{u}_k \tag{1.20}$$

2 | Surrogate Model via FNO

2.1. Generalities on surrogate models

Surrogate models are simplified analytical representations designed to emulate the input/output behaviour of intricate systems [9]. Constructing these models necessitates conducting computationally intensive simulations at specific, meticulously chosen sample points. These surrogate models approximate the behavior of the underlying complex simulations with a reasonable level of precision, all the while offering a more computationally economical means of evaluation.

The development of surrogate models involves the utilization of computer experiments, wherein design parameters span a thoughtfully selected range within the design space. These parameters adhere to predefined specifications or patterns, as guided by a technique known as the design of experiments. The computationally intensive simulations are executed at these selected points, recording their corresponding responses. Using this input/output data, surrogate models emerge as straightforward functional approximations, effectively capturing the complex simulation's behavior. Consequently, the intricate details of the complex simulation become less critical, with the surrogate model taking the spotlight. These surrogate models can be grounded in equations that simplify complex systems by omitting higher-order terms or by assuming lumped systems instead of distributed ones [16].

The application of surrogate models is particularly apparent in multi-scale modeling. Historically, modeling often focused on a specific scale within a system, while disregarding the impact of other scales through assumptions or constitutive relations. In contrast, multi-scale modeling encompasses a comprehensive view, simultaneously considering various scales within a system and combining them into an overarching model. The goal is to harness the efficiency of a higher scale while maintaining the accuracy of a lower scale. In this context, the interactions between different scales significantly influence the quality of the overall model.

2.2. Neural Operators

Numerous challenges in science and engineering involve the recurrent solving of intricate systems of partial differential equations (PDEs) across a range of parameter values. This situation arises in fields like molecular dynamics, micro-mechanics, and the modeling of turbulent flows. Often, these systems necessitate fine-grained discretization to accurately depict the phenomena under investigation. Consequently, traditional numerical solvers tend to be slow and sometimes inefficient.

Machine learning techniques offer expedited solvers that approximate or augment traditional approaches [10, 12, 15, 26]. However, classical neural networks operate within finite-dimensional spaces, which restricts them to learning solutions associated with a specific discretization. This limitation frequently hampers practical applications, necessitating the development of mesh-invariant neural networks. In this context, two prominent neural network-based strategies for tackling PDEs has been historically adopted [18]: finite-dimensional operators and Neural-FEM.

1. **Finite-Dimensional Operators:** These methods represent the solution operator using deep convolutional neural networks within finite-dimensional Euclidean spaces [1, 4, 11, 14, 29]. As the name implies, these approaches depend on the mesh's characteristics and require adjustments and fine-tuning for varying resolutions and discretizations to achieve consistent error. Moreover, their applicability is confined to the discretization size and geometry of the training data, making it impossible to extrapolate solutions to new points in the domain.

2. **Neural-FEM:** This approach directly parameterizes the solution function as a neural network [2, 8, 24, 26, 27]. Unlike the first approach, it models a specific instance of the PDE rather than the solution operator. It is mesh-independent and accurate but necessitates training a new neural network for each new instance of the functional parameter or coefficient. It closely resembles classical methods like finite elements, replacing the linear span of a finite set of local basis functions with the space of neural networks. However, like traditional methods, it faces computational challenges as the optimization problem must be solved anew for each new instance, and it assumes knowledge of the underlying PDE.

Following [18], a third approach can be considered: **Neural Operators**. The emergence of Neural Operators represents a novel approach aimed at learning mesh-free, infinite-dimensional operators using neural networks [5, 17, 19, 23, 25]. The neural operator mitigates the mesh-dependent nature of finite-dimensional operator methods by produc-

ing a single set of network parameters usable with various discretizations. It can efficiently transfer solutions between meshes and only requires one-time training. Obtaining a solution for a new parameter instance involves a straightforward forward pass of the network, reducing the computational overhead seen in Neural-FEM methods. Most notably, the neural operator operates without any prior knowledge of the underlying PDE, relying solely on data. However, to date, neural operators have not produced efficient numerical algorithms comparable to the success of convolutional or recurrent neural networks in the finite-dimensional setting due to the cost of evaluating integral operators. Our work addresses this challenge through the utilization of the fast Fourier transform.

2.3. Learning operators

The presented methodology involves learning a mapping between two infinite-dimensional spaces, using a finite collection of observed input-output pairs. Let's consider $D \in \mathbb{R}^d$ as a bounded, open set, and $\mathcal{A} = \mathcal{A}(D; \mathbb{R}^{d_a})$ and $\mathcal{U} = \mathcal{U}(D; \mathbb{R}^{d_u})$ as separable Banach spaces of functions that take values in $\mathbb{R}^{d_a}$ and $\mathbb{R}^{d_u}$, respectively. In a general case, it is typical to deal with a nonlinear mapping $G^\dagger : \mathcal{A} \to \mathcal{U}$, which often arises as the solution operator of parametric PDEs. Let assume to have at disposal observations $\{a_j, u_j\}_{j=1}^N$, where $a_j \sim \mu$ is an independent and identically distributed sequence sampled from the probability measure μ supported on $\mathcal{A}$, and $u_j = G^\dagger(a_j)$, which may be potentially corrupted by noise. Our goal is to construct an approximation of $G^\dagger$ by creating a parametric map

$$G : \mathcal{A} \times \Theta \to \mathcal{U}$$

or equivalently

$$G_\theta : \mathcal{A} \to \mathcal{U}, \text{with } \theta \in \Theta$$

being Θ a finite-dimensional parameter space, by selecting $\theta^\dagger$ so that $G(\cdot, \theta^\dagger) = G_{\theta^\dagger} \approx G^\dagger$. This framework is well-suited for learning in infinite dimensions and aligns with the classic finite-dimensional setting. Defining an appropriate cost functional

$$C : \mathcal{U} \times \mathcal{U} \to \mathbb{R}$$

the problem reduces to find its minimum:

$$\min_{\theta \in \Theta} \mathbb{E}_{a \sim \mu}[C(G(a, \theta), G^\dagger(a))]$$

This framework mirrors the classical finite-dimensional setting [28]. Demonstrating the

existence of minimizers in the infinite-dimensional setting remains a challenging open problem. To address this, the problem can be approached in the test-train setting, using data-driven empirical approximations to the cost to determine θ and test the accuracy of the approximation. Since our methodology is framed in the infinite-dimensional context, all finite-dimensional approximations share a common set of parameters that remain consistent in infinite dimensions.

Approximating the operator $G^\dagger$ is a distinct and typically more challenging task than finding the solution $u \in \mathcal{U}$ for a single instance of the parameter $a \in \mathcal{A}$. Existing methods, (including classical finite elements, finite differences, and finite volumes, as well as modern machine learning techniques like physics-informed neural networks [26]), primarily focus on the latter, which can be computationally expensive. In contrast, the approach here presented directly approximates the operator, making it more cost-effective and faster, offering significant computational savings compared to traditional solvers.

Since data a_j and u_j are generally functions, one can only work with point-wise evaluations of them. Consider $D_j = \{x_1, ..., x_n\} \subset D$ to be an n-point discretization of the domain D, and assume to have at disposal observations $a_j|_{D_j} \in \mathbb{R}^{n \times d_a}$ and $u_j|_{D_j} \in \mathbb{R}^{n \times d_u}$, forming a finite collection of input-output pairs indexed by j. To be discretization-invariant, the neural operator can provide an output $u(x)$ for any $x \in D$, even for $x \notin Dj$. This property is highly desirable as it allows for the transfer of solutions between different grid geometries and discretizations.

Following [17], the Neural Operator can be formulated as an iterative architecture

$$v_0 \to v_1 \to ... \to v_T$$

, where v_j for $j = 0, 1, ..., T-1$ is a sequence of functions taking values in $\mathbb{R}^{d_v}$. As illustrated in Figure 2.1, the input $a \in \mathcal{A}$ is initially transformed into a higher-dimensional representation $v_0(x) = P(a(x))$ using the local transformation P, typically parameterized by a shallow fully-connected neural network (dense layer). Subsequently, several iterations of updates $v_t \to v_{t+1}$ are applied. The output $u(x) = Q(v_T(x))$ is the projection of v_T by the local transformation Q: $\mathbb{R}^{d_v} \to \mathbb{R}^{d_u}$.

In each iteration, the update $v_t \to v_{t+1}$ is defined as the composition of a non-local integral operator $\mathcal{K}$ and a local, nonlinear activation function σ. Following this scheme, two definitions can be made.

Iterative updates:

$$v_{t+1}(x) := \sigma(W v_t(x) + (\mathcal{K}(a;\phi)v_t)(x)) \qquad \forall x \in D \tag{2.1}$$

In 2.1, $\mathcal{K} : \mathcal{A} \times \Theta_{\mathcal{K}} \to \mathcal{L}(\mathcal{U}(D;\mathbb{R}^{d_v}),\mathcal{U}(D;\mathbb{R}^{d_v}))$ is a map to bounded linear operators $\mathcal{L}$ defined on $\mathcal{U}(D;\mathbb{R}^{d_v})$ and it is parameterized by $\phi \in \Theta_{\mathcal{K}}$, while $W : \mathbb{R}^{d_v} \to \mathbb{R}^{d_v}$ is a linear transformation and σ is a non linear activation function.

Kernel integral operator:

$$(\mathcal{K}(a;\phi)v_t)(x) := \int_D \kappa_\phi(x, y, a(x), a(y))v_t(y)\, dy \qquad \forall x \in D \tag{2.2}$$

where $\kappa_\phi : \mathbb{R}^{2(d+d_a)} \to \mathbb{R}^{d_v \times d_v}$ is a neural network parameterized by $\phi \in \Theta_{\mathcal{K}}$.

Equations 2.1 and 2.2 together define the generalization of neural network to the infinite dimensional framework. Note that even if the integral operator is linear, highly non linear operator can be learned thanks to the non-linearity introduced by the activation function [17].

2.4. Fourier Neural Operators

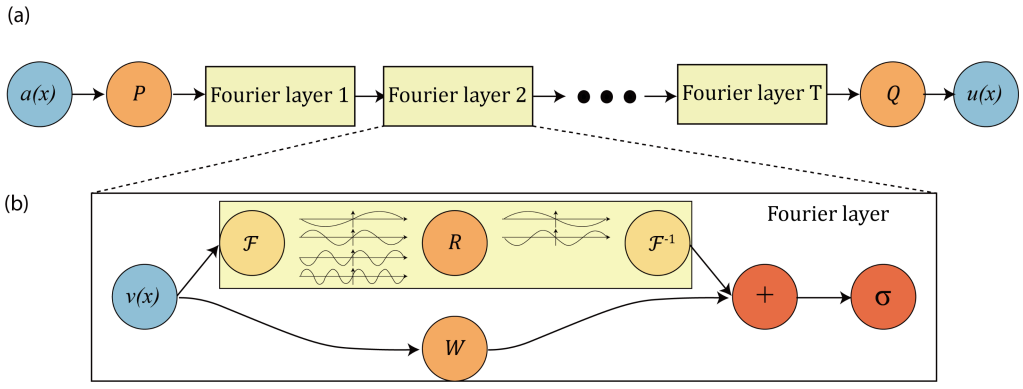


Figure 2.1: Full architecture of a Fourier Neural Operator (a) and the Fourier Layer (b)

This section deals with a Neural Operator in which the kernel integral operator is a convolution operator defined in the Fourier space [18]. Considering the following definition as a particular case to be used in 2.2:

$$\kappa_\phi(x, y, a(x), a(y)) = \kappa_\phi(x - y) \qquad \forall x \in D, \tag{2.3}$$

then exploiting the convolution theorem it is possible to write the kernel integral operator as:

$$(\mathcal{K}(a;\phi)v_t)(x) = \mathcal{F}^{-1}(\mathcal{F}(\kappa_\phi)\cdot\mathcal{F}(v_t))(x) \qquad \forall x \in D. \tag{2.4}$$

It is possible to take κ_ϕ directly in the Fourier space so that 2.4 becomes:

$$(\mathcal{K}(a;\phi)v_t)(x) = \mathcal{F}^{-1}(R_\phi\cdot\mathcal{F}(v_t))(x) \qquad \forall x \in D, \tag{2.5}$$

where R_ϕ is the Fourier transform of some periodic function $\kappa : \bar{D} \to \mathbb{R}^{d_v\times d_v}$ parameterized by $\phi \in \Theta_\mathcal{K}$. In practice R_ϕ can be seen as a linear transformation to be applied in the Fourier space. It is indeed a complex-valued tensor with dimensions equal to $k_{max}\times d_v\times d_v$ to be applied to the Fourier transform of v_t, where k_{max} is the highest frequency mode one chose to keep. In principle any choice of k_{max} could be taken, but in practice this value is bounded by the dimension of v_t as a consequence of the definition of the Fast Fourier Transform algorithm (FFT) as shown in section 2.5.

2.5. Discretization and FFT

If the domain D is discretized with $n \in \mathcal{N}$ points, then $v_t \in \mathbb{R}^{n\times v_d}$ and $\mathcal{F}(v_t) \in \mathbb{C}^{n\times d_v}$. Since the convolution with R has to consider only the first $k_{max} < n$ modes, it is necessary to truncate the Fourier expansion of v_t, so that $\mathcal{F}(v_t) \in \mathbb{C}^{k_{max}\times d_v}$

If the discretization is uniform across all the dimension, let say with resolution $s_1\times...\times s_d = n$, then the Fourier transform $\mathcal{F}$ can be efficiently replaced by the FFT $\hat{\mathcal{F}}$ (and analogously for its inverse). Given a function $f \in \mathbb{R}^{n\times d_v}$, and being $k = (k_1,...,k_d) \in \mathbb{Z}_{s_1}\times...\times\mathbb{Z}_{s_d}$ and $x = (x_1,...,x_d) \in D$, the FFT and its inverse are defined as:

$$(\hat{\mathcal{F}}f)_l(k) = \sum_{x_1=0}^{s_1-1}\cdots\sum_{x_d=0}^{s_d-1} f_l(x_1,...,x_d)e^{-2i\pi\sum_{j=1}^{d}\frac{x_jk_j}{s_j}} \qquad l = 1,...,d_v \tag{2.6}$$

$$(\hat{\mathcal{F}}^{-1}f)_l(x) = \sum_{k_1=0}^{s_1-1}\cdots\sum_{k_d=0}^{s_d-1} f_l(k_1,...,k_d)e^{2i\pi\sum_{j=1}^{d}\frac{x_jk_j}{s_j}} \qquad l = 1,...,d_v \tag{2.7}$$

Although the maximum number of modes to be considered depends on the discretization, The Fourier layers maintain their discretization-invariance because they have the capacity to learn from and assess functions regardless of how they are discretized. As the model learns parameters directly in Fourier space, the process of representing these functions in physical space is essentially a matter of projecting them onto the basis $e^{2\pi\langle x\cdot k\rangle}$, which

remains well-defined across the entirety of $\mathbb{R}^d$.

More details about the implementation of an actual FNO in Python can be found in chapter 4.

3 | Implementation of EnKF model

In this chapter it is explained implementation of the data assimilation procedure and also for the Ensemble Kalman filter method in Python. There have been developed two different classes; in particular the last one is a child class of the more general data assimilation class.

3.1. Libraries and dependencies

The code developed takes advantage of the libraries presented below. In particular arrays and matrices are worked out by exploiting NumPy library. The creation of abstract base classes in Python is performed by the abc module. The Kalman filtering is implemented thanks to FilterPy which is a Python library that implements a number of Kalman filters and that is then employed to handle the complexities of the Ensemble Kalman Filter algorithm.

```
1     import numpy as np
2     from abc import ABC
3     from filterpy.kalman import EnsembleKalmanFilter as EnKF_model
```

3.2. Data Assimilation class structure

Complete code for the class

```
1 class Data_Assimilation(ABC):
2     def __init__(self, dim_x, dim_z, f, h, get_data, dt=1, t0=0):
3         self.dim_x = dim_x
4         self.dim_z = dim_z
5         self.dt = dt
6         self.get_data = get_data
7         self.x = np.zeros((dim_x,))
8         self.z = np.zeros((dim_z,))
```

```
        self.f = f
        self.h = h
        self.t0 = t0
        self.t = t0
        self.model = None
```

This class is made of a constructor that provides a generic structure for data assimilation algorithms. It initializes the data assimilation system with essential parameters:

- `dim_x`: Integer representing the dimension of the state variables,
- `dim_z` : Integer indicating the dimension of the observational data,
- `f` : Function defining the state transition dynamics,
- `h` : Function representing the measurement process,
- `get_data`: Function for acquiring observed data,
- `dt` : Time step size, which is taken by default equal to 1,
- `t0` : Initial time, taken by default as zero.

State variables and data are initialized respectively as zero vectors of dimensions `dim_x` and `dim_z`, the current time t is initialized to the initial time and there is a placeholder, called "model", which is needed for the specific data assimilation mode defined in the subclasses.

3.3. EnKF class structure

Complete code for the class:

```
class EnKF(Data_Assimilation):
    def create_model(self, x0, P, R, Q, N=1000):
        self.model = EnKF_model(x=x0, P=P, dim_z=self.dim_z, dt=self.dt,
    N=N,hx=self.h, fx=self.f)
        self.model.R = R
        self.model.Q = Q

    def predict(self):
        self.model.predict()

    def update(self, z):
        self.t += self.dt
        self.model.update(z)

```

```
    def predict_and_update(self):
        self.z = self.get_data(self.t)
        self.predict()
        self.update(self.z)

    def loop(self, T, verbose=False):
        Nt = np.int32((T-self.t0)/self.dt)
        x_hat = np.zeros((Nt+1, self.dim_x))
        x_hat[0,:] = self.model.x
        if  self.t >= T:
            raise "Current time is {} that is less or equal to end time
   {}".format(self.t, T)
        for i in range(Nt):
            if verbose:
                print('Advancing: ' + str(i/Nt*100) + '%')
            self.predict_and_update()
            x_hat [i+1,:] = self.model.x
        return x_hat
```

This is a child class of `Data_Assimilation` and for this reason an object of EnKf has the same members of the general class. There could have been other child classes representing different data assimilation models. In this case the only child class implemented is this one, which extends the general one implementing the Ensemble Kalman Filter model.

Create model The first step implemented is the creation of the actual Ensemble Kalman Filter model by exploiting `filterpy.kalman.EnsembleKalmanFilter` which is an already existing library. To create it are necessary more input data, which are:

- `x0`: It is the initial state vector,
- `P`: Initial estimate of state covariance,
- `R`: Measurement noise covariance,
- `Q`: Process noise covariance,
- `N`: Number of ensemble members, taken by default as 1000.

```
def create_model(self, x0, P, R, Q, N=1000):
        self.model = EnKF_model(x=x0, P=P, dim_z=self.dim_z, dt=self.dt,
    N=N,hx=self.h, fx=self.f)
        self.model.R = R
        self.model.Q = Q
```

Already existing methods The `predict` method is an already existing method, given by `filterpy.kalman.EnsembleKalmanFilter` which advances the state estimation without incorporating new observational data. Also the `update` method is given by the same library and updates the state estimation based on new observational data `z` which is given as input. These two methods are combined together in `predict_and_update` function.

```
def predict(self):
        self.model.predict()

    def update(self, z):
        self.t += self.dt
        self.model.update(z)

    def predict_and_update(self):
        self.z = self.get_data(self.t)
        self.predict()
        self.update(self.z)
```

Loop This function executes the function `predict_and_update` in a specific time range which depends on the final time `T` given as input. This function returns the estimated state over time `x_hat`. This function takes as input also the term verbose, taken by default as false, if it is true the function displays the time progress made.

```
def loop(self, T, verbose=False):
        Nt = np.int32((T-self.t0)/self.dt)
        x_hat = np.zeros((Nt+1, self.dim_x))
        x_hat[0,:] = self.model.x
        if  self.t >= T:
            raise "Current time is {} that is less or equal to end time
  {}".format(self.t, T)
        for i in range(Nt):
            if verbose:
                print('Advancing: ' + str(i/Nt*100) + '%')
            self.predict_and_update()
            x_hat [i+1,:] = self.model.x
        return x_hat
```

4 | Implementation of FNO surrogate model

The aim of this chapter is to present the details of the actual implementation of a Fourier Neural Operator in Python. Two set of classes has been developed, one for mono-dimensional problems (see section 4.2) and one for two-dimensional problems (see section 4.3). The two sets follow the same structure, but when dealing with multi-dimensional problems some small adjustments in the tensors shape is required.

4.1. Libraries and dependencies

The code developed is based on some common libraries. In particular, array and tensor are worked out by exploiting NumPy and TensorFlow libraries. The neural network architecture has been developed using `keras` custom layers. In the following the code to import all the libraries needed in order for the class to work properly.

```
1    import numpy as np
2    import tensorflow as tf
```

4.2. One-Dimensional FNO classes structure

These classes are made for dealing with one-dimensional problems, meaning that input and output of the FNO have to be represented by one-dimensional arrays. In order to reproduce the architecture depicted in figure 2.1, three classes have been developed. The first represent the FFT-Layer that computes the convolutions, the second serves as a bias layer, finally the third combine these two layer to obtain the complete Fourier layer. A FNO is made of several blocks formed by Fourier layer, eventually preceded and followed by linear transformations that elevate the input to an higher dimension and bring it back to the output dimension after the several convolutions. This architecture can be generated by a useful function `FNO` that give to the user the possibility to easily create a FNO with a custom size, but the three sub-class could also be assembled in a custom fashion.

These three classes are constructed as derived class from the class `tf.keras.layers.Layer` and they exploit useful python inheritance properties. As child classes, they inherit all the properties and methods from the parent class, with the addition of specific properties or methods The decorator `@tf.keras.utils.register_keras_serializable()` in the first line of all the definitions is needed to correctly save to file and then load `keras` models.

4.2.1. The FFT-Layer class

Complete code for the class:

```
@tf.keras.utils.register_keras_serializable()
class FFT_Layer(tf.keras.layers.Layer):
    def __init__(self, k_max=None, **kwargs):
        super(FFT_Layer, self).__init__(**kwargs)
        self._fft_shape = None
        self._ifft_shape = None
        self.k_max = k_max

    def build(self, input_shape):
        if self.k_max == None:
            self._fft_shape = tf.convert_to_tensor(input_shape[-1] // 2
  + 1, dtype=tf.int32)
            self._ifft_shape = tf.multiply(tf.convert_to_tensor(
  input_shape[-1] // 2, dtype=tf.int32), 2)
        else:
            self._fft_shape = tf.convert_to_tensor(self.k_max, dtype=tf.
  int32)
            self._ifft_shape = tf.multiply(tf.convert_to_tensor(self.
  k_max-1, dtype=tf.int32), 2)
        print('fft_shape set:', self._fft_shape.numpy())
        print('ifft_shape set:', self._ifft_shape.numpy())

        self.kernel = self.add_weight(
            name="kernel",
            shape=(self.fft_shape, self._fft_shape),
            initializer="glorot_uniform",
            trainable=True
        )

    def call(self, inputs):
        fft = tf.signal.rfft(inputs)
        if not(self.k_max==None):
            fft = fft[..., :self.k_max]
```

```
        kernel_complex = tf.complex(self.kernel, tf.zeros_like(self.
    kernel))
        r = tf.linalg.matmul(fft, kernel_complex)
        ifft = tf.signal.irfft(r)
        return ifft

    def get_config(self):
        config = super().get_config()
        config["k_max"] = self.k_max
        return config

    @property
    def fft_shape(self):
        return self._fft_shape

    @property
    def ifft_shape(self):
        return self._ifft_shape
```

This class produces a `keras` custom layer and it basically computes the FFT of the input (eventually truncated up to `k_max`), it applies a linear transformation and finally compute the inverse FFT. In the following, all the methods are presented.

Constructor It is specified by the Python command `__init__` and it takes as arguments the maximum number of modes to be considered (`None` by default, i.e. all modes) and any eventual keyword arguments. It simply calls the parent class constructor and initializes class members `self._fft_shape` and `self._ifft_shape` to `None`, while setting `self.k_max = k_max` to the specified value. The shapes involved in the Fourier transform and its inverse depend on the size of the input array and they are potentially not known at the moment of the definition of an object of this class, so that their initialization to `None` is necessary.

```
def __init__(self, k_max=None, **kwargs):
    super(FFT_Layer, self).__init__(**kwargs)
    self._fft_shape = None  # Shape after FFT
    self._ifft_shape = None # Shape after IFFT
    self.k_max = k_max
```

Building the layer Given a shape of the input array `input_shape`, the `build` method creates the actual tensors in the layer. The values of `self._fft_shape` and `self._ifft_shape` are derived from the shape of the input or eventually set to be equal to `k_max`. Recall that

given a real valued array with $2N$ entries, its FFT has a number of independent component equal to $N+1$. After deducing the shapes of tensors after the application of the FFT and the inverse FFT, this method creates a linear transformation tensor in the Fourier domain, here denoted as `self.kernel`, through the parent class method `add_weight`.

```
def build(self, input_shape):
    if self.k_max == None:
        self._fft_shape = tf.convert_to_tensor(input_shape[-1] // 2 + 1,
    dtype=tf.int32)
        self._ifft_shape = tf.multiply(tf.convert_to_tensor(input_shape
    [-1] // 2, dtype=tf.int32), 2)
    else:
        self._fft_shape = tf.convert_to_tensor(self.k_max, dtype=tf.
    int32)
        self._ifft_shape = tf.multiply(tf.convert_to_tensor(self.k_max
    -1, dtype=tf.int32), 2)
    print('fft_shape set:', self._fft_shape.numpy())
    print('ifft_shape set:', self._ifft_shape.numpy())

    self.kernel = self.add_weight(
        name="kernel",
        shape=(self.fft_shape, self._fft_shape),
        initializer="glorot_uniform",
        trainable=True
    )
```

Calling the layer The call method actually perform the computations on given inputs. Firstly it compute the FFT (eventually truncated up to `k_max` modes), then it apply the linear transformation in the Fourier space (where tensor are complex-valued) and finally it returns the inverse FFT of the result of the latter.

```
def call(self, inputs):
    fft = tf.signal.rfft(inputs)
    if not(self.k_max==None):
        fft = fft[..., :self.k_max]
    kernel_complex = tf.complex(self.kernel, tf.zeros_like(self.kernel))
    r = tf.linalg.matmul(fft, kernel_complex)
    ifft = tf.signal.irfft(r)
    return ifft
```

Specifying custom inputs `keras` present useful methods for saving and loading models and these methods work fine for architecture made of standard layers, but whenever a custom layer is defined, a list of the custom inputs to the constructor has to be provided,

together with the already motioned decorator. The method `get_config` accomplish to this task by adding the new input to the input list, here denoted by `config`.

```
def get_config(self):
    config = super().get_config()
    config["k_max"] = self.k_max
    return config
```

Properties The structure of the Bias-Layer depicted in 4.2.2 depends on the shape of the tensor that comes as output from the inverse FFT. For having this information outside the class, two properties regarding the tensor shape in the Fourier convolution process have been added. These properties can also be useful for a custom arrangement of the defined custom layers.

```
@property
def fft_shape(self):
    return self._fft_shape

@property
def ifft_shape(self):
    return self._ifft_shape
```

4.2.2. The Bias-Layer class

Complete code for the class:

```
@tf.keras.utils.register_keras_serializable()
class Bias_Layer(tf.keras.layers.Layer):
    def __init__(self, fft_layer_object, **kwargs):
        super(Bias_Layer, self).__init__(**kwargs)
        self.fft_layer_object = fft_layer_object

    def build(self, input_shape):
        self.kernel = self.add_weight(
            name="kernel",
            shape=(input_shape[-1], self.fft_layer_object.ifft_shape),
            initializer="glorot_uniform",
            trainable=True
        )
        print('Bias layer has shape: '+str(self.fft_layer_object.
    ifft_shape.numpy()))

    def call(self, inputs):
        bias = tf.linalg.matmul(inputs, self.kernel)
```

```
18         return bias
19
20     def get_config(self):
21         config = super().get_config()
22         config["fft_layer_object"] = self.fft_layer_object
23         return config
```

This class creates a `keras` custom layer that essentially perform a linear transformation to the input vector. In the following, all the methods are presented.

Constructor It is specified by the Python command `__init__` and it takes as argument an object of the class `FFT_Layer`. It calls the parent class constructor and store the specified `FFT_Layer` object as the member `self.fft_layer_objetc`

```
def __init__(self, fft_layer_object, **kwargs):
    super(Bias_Layer, self).__init__(**kwargs)
    self.fft_layer_object = fft_layer_object
```

Building the layer Given a shape of the input array `input_shape`, the build method creates the actual tensors in the layer. In this layer the only tensor is the linear transformation tensor, here denoted as `self.kernel`, created by means of the parent class method `add_weight`.

```
def build(self, input_shape):
    self.kernel = self.add_weight(
        name="kernel",
        shape=(input_shape[-1], self.fft_layer_object.ifft_shape),
        initializer="glorot_uniform",
        trainable=True
    )
    print('Bias layer has shape: '+str(self.fft_layer_object.ifft_shape.
   numpy()))
```

Calling the layer The call methods actually perform the computations on given inputs. For this layer it simply perform a multiplication between the input and the tensor `self.kernel`

```
def call(self, inputs):
    bias = tf.linalg.matmul(inputs, self.kernel)
    return bias
```

Specifying custom inputs As already seen, for a correct saving and loading of custom layers in `keras`, it is essential to specify the inputs of the custom layers. This is done in the method `get_config`

```
def get_config(self):
    config = super().get_config()
    config["fft_layer_object"] = self.fft_layer_object
    return config
```

4.2.3. The Fourier-Layer class

Complete code for the class:

```
@tf.keras.utils.register_keras_serializable()
class Fourier_Layer(tf.keras.layers.Layer):
    def __init__(self, k_max=None, **kwargs):
        super(Fourier_Layer, self).__init__(**kwargs)
        self.fft_layer = FFT_Layer(k_max=k_max)
        self.bias_layer = Bias_Layer(self.fft_layer)
        self.k_max = k_max

    def call(self, inputs):
        fft_layer = self.fft_layer(inputs)
        bias_layer = self.bias_layer(inputs)
        added_layers = layers.Add() ([fft_layer, bias_layer])
        return layers.Activation('relu') (added_layers)

    def get_config(self):
        config = super().get_config()
        config["k_max"] = self.k_max
        return config
```

This class call the previous two classes for creating a FFT-Layer and a Bias-Layer, then performs the sum between the outputs of these layers and finally it applies a non-linear activation function. Recall that thanks to the fact that the Bias-Layer is created after the FFT-Layer, it is aware of the correct shape to give to the tensors so that they can be suitable for a sum. Since this class does not need to create any tensor, it does not require a `build` method.

Constructor It is specified by the Python command `__init__` and it takes as argument the number of modes to keep in the Fourier space. It calls the parent class constructor, then it initialize two layers of type `FFT_Layer` and `Bias_Layer`, finally it store the desired number of modes in a member variable `self.k_max`.

```
def __init__(self, k_max=None, **kwargs):
    super(Fourier_Layer, self).__init__(**kwargs)
    self.fft_layer = FFT_Layer(k_max=k_max)
    self.bias_layer = Bias_Layer(self.fft_layer)
    self.k_max = k_max
```

Calling the layer The call methods actually perform the computations on given inputs. In this case it performs a sum of the tensor coming from the FFT-Layer and the Bias-Layer and it applies a non-linear activation function to the result. Since all the other computations are essentially linear transformation, this is the key point in the model where non-linearity arises.

```
def call(self, inputs):
    fft_layer = self.fft_layer(inputs)
    bias_layer = self.bias_layer(inputs)
    added_layers = layers.Add() ([fft_layer, bias_layer])
    return layers.Activation('relu') (added_layers)
```

Specifying custom inputs Also in this case a list of inputs of the custom layer has to be provided. This is accomplished by the method `get_config`.

```
def get_config(self):
    config = super().get_config()
    config["k_max"] = self.k_max
    return config
```

4.2.4. The FNO function

Function definition:

```
def FNO(INPUTDIM, OUTPUTDIM, p_dim, n, k_max=None, verbose=False,
   model_name='FNO', dropout=0.0, kernel_reg=0.0):
    input_layer = layers.Input(shape = INPUTDIM, name= 'input_layer')
    P_layer = layers.Dense(p_dim, activation='relu', kernel_regularizer
   = regularizers.l2(kernel_reg), name='P_layer') (input_layer)
    P_layer = layers.Dropout(dropout) (P_layer)
    # Repeat the custom module 'n' times
    for i in range(n):
        if verbose:
            print('Creating Fourier Layer ' +str(i))
        if i ==0:
            fourier_module_output = Fourier_Layer(name='fourier_layer_'+
   str(i), k_max=k_max)(P_layer)
```

```
        else:
            fourier_module_output = Fourier_Layer(name='fourier_layer_'+
    str(i), k_max=k_max)(fourier_module_output)
     output_layer = layers.Dense(OUTPUTDIM[0], activation='linear',
    kernel_regularizer = regularizers.l2(kernel_reg), name='output_layer'
    ) (fourier_module_output)
     output_layer= layers.Dropout(dropout) (output_layer)
     if verbose:
         print('----------------------------------------------------------')
     model = tf.keras.Model(inputs=input_layer, outputs = output_layer,
    name = model_name)
     if verbose:
         model.summary()
     return model
```

This function automatically creates a FNO 1D structure of the proper dimension. Variables `INPUTDIM` and `OUTPUTDIM` are the input and output shapes, they only determine the shapes respectively of the input layer and the output layer. `P_dim` is the shape of the so-called P-layer, that is a dense layer that elevates the input to a higher dimensionality. The variables `n` and `k_max` are respectively the number of Fourier Layer to be concatenated an the maximum number of modes to be kept in each Fourier convolution. For improving training, it is possible to assign an optional dropout or kernel regularization to the P-layer and the output layer by means of the optional parameters `dropout` and `kernel_reg`. The Boolean parameter `verbose` determine if some messages will be displayed during the creation of the FNO, while `model_name` is an optional name to the model.

4.3. Two-Dimensional FNO classes structure

In the 2D case, inputs and outputs are tensor of order 2 and this require some attention to be paid regarding tensor shapes and layer definition. To be meaningful, the FFT has to be performed on a tensor of the same order of the input, this means that the two-dimensional input cannot simply be flattened. Nevertheless, a non flattened tensor cannot be elevated to an higher dimension trough a dense layer. The simplest solution to this issue is to first flatten the input tensor in a mono-dimensional array, so that it can be passed to a dense P-layer, and then reshaping it to a two-dimensional tensor for entering the Fourier Layers. The same approach is performed also on the output layer, only to be able to properly apply eventual dropouts or regularizations.

Except for these small adjustments, the logical structure of the classes is the same, and

therefore the code is reported as a whole, without a detailed treatment.

```
@tf.keras.utils.register_keras_serializable()
class FFT_Layer_2D(tf.keras.layers.Layer):
    def __init__(self, k_max=None, **kwargs):
        super(FFT_Layer_2D, self).__init__(**kwargs)
        self._fft_shape = None
        self._ifft_shape = None
        self.k_max = k_max

    def build(self, input_shape):
        if self.k_max == None:
            self._fft_shape = tf.convert_to_tensor(input_shape[-1] // 2
    + 1, dtype=tf.int32)
            self._ifft_shape = tf.multiply(tf.convert_to_tensor(
    input_shape[-1] // 2, dtype=tf.int32), 2)
        else:
            self._fft_shape = tf.convert_to_tensor(self.k_max, dtype=tf.
    int32)
            self._ifft_shape = tf.multiply(tf.convert_to_tensor(self.
    k_max-1, dtype=tf.int32), 2)
        print('fft_shape set:', self._fft_shape.numpy())
        print('ifft_shape set:', self._ifft_shape.numpy())

        self.kernel = self.add_weight(
            name="kernel",
            shape=(self._fft_shape, self._fft_shape),
            initializer="glorot_uniform",
            trainable=True
        )

    def call(self, inputs):
        fft = tf.signal.rfft2d(inputs)
        if not(self.k_max==None):
            fft = fft[..., :self.k_max]
        kernel_complex = tf.complex(self.kernel, tf.zeros_like(self.
    kernel))
        r = tf.linalg.matmul(fft, kernel_complex)
        ifft = tf.signal.irfft2d(r)
        return ifft

    def get_config(self):
        config = super().get_config()
        config["k_max"] = self.k_max
        return config
```

```
39
40      @property
41      def fft_shape(self):
42          return self._fft_shape
43
44      @property
45      def ifft_shape(self):
46          return self._ifft_shape
47
48  @tf.keras.utils.register_keras_serializable()
49  class Bias_Layer_2D(tf.keras.layers.Layer):
50      def __init__(self, fft_layer_object, **kwargs):
51          super(Bias_Layer_2D, self).__init__(**kwargs)
52          self.fft_layer_object = fft_layer_object
53
54      def build(self, input_shape):
55          self.kernel = self.add_weight(
56              name="kernel",
57              shape=(input_shape[-1], self.fft_layer_object.ifft_shape),
58              initializer="glorot_uniform",
59              trainable=True
60          )
61          print('Bias layer has shape: '+str(self.fft_layer_object.
    ifft_shape.numpy()))
62
63      def call(self, inputs):
64          bias = tf.linalg.matmul(inputs, self.kernel)
65          return bias
66
67      def get_config(self):
68          config = super().get_config()
69          config["fft_layer_object"] = self.fft_layer_object
70          return config
71
72  @tf.keras.utils.register_keras_serializable()
73  class Fourier_Layer_2D(tf.keras.layers.Layer):
74      def __init__(self, k_max=None, **kwargs):
75          super(Fourier_Layer_2D, self).__init__(**kwargs)
76          self.fft_layer = FFT_Layer_2D(k_max=k_max)
77          self.bias_layer = Bias_Layer_2D(self.fft_layer)
78          self.k_max = k_max
79
80      def call(self, inputs):
81          fft_layer = self.fft_layer(inputs)
```

```
82          bias_layer = self.bias_layer(inputs)
83          added_layers = layers.Add() ([fft_layer, bias_layer])
84          return layers.Activation('relu') (added_layers)
85
86      def get_config(self):
87          config = super().get_config()
88          config["k_max"] = self.k_max
89          return config
90
91  def FNO2D(INPUTDIM, OUTPUTDIM, p_dim, n, k_max=None, verbose=False,
        model_name='FNO2D', dropout=0.0, kernel_reg=0.0):
92      input_layer = layers.Input(shape = INPUTDIM, name= 'input_layer')
93      input_layer_flat = layers.Reshape((INPUTDIM[0]*INPUTDIM[1],)) (
        input_layer)
94      P_layer = layers.Dense(p_dim**2, activation='relu',
        kernel_regularizer = regularizers.l2(kernel_reg), name='P_layer') (
        input_layer_flat)
95      P_layer = layers.Dropout(dropout) (P_layer)
96      P_layer = layers.Reshape((p_dim, p_dim)) (P_layer)
97      # Repeat the custom module 'n' times
98      for i in range(n):
99          if verbose:
100             print('Creating Fourier Layer ' +str(i))
101         if i ==0:
102             fourier_module_output = Fourier_Layer_2D(name='
        fourier_layer_'+str(i), k_max=k_max)(P_layer)
103         else:
104             fourier_module_output = Fourier_Layer_2D(name='
        fourier_layer_'+str(i), k_max=k_max)(fourier_module_output)
105     fourier_module_output = layers.Reshape((p_dim*2*(k_max-1),)) (
        fourier_module_output)
106     output_layer_flat = layers.Dense(OUTPUTDIM[0]*OUTPUTDIM[1],
        activation='linear', kernel_regularizer = regularizers.l2(kernel_reg)
        , name='output_layer') (fourier_module_output)
107     output_layer_flat = layers.Dropout(dropout) (output_layer_flat)
108     output_layer = layers.Reshape(OUTPUTDIM) (output_layer_flat)
109     if verbose:
110         print('----------------------------------------------------------')
111     model = tf.keras.Model(inputs=input_layer, outputs = output_layer,
        name = model_name)
112     if verbose:
113         model.summary()
114     return model
```

5 | EnKF numerical examples

In this chapter are presented the numerical examples that have ben developed in order to verify that `Data assimilation` and `EnKF` work correctly. For this reason we consider the Prey-Predator model developed for different scenarios. It will be first introduced what is intended by Prey-Predator model and its mathematical formulation and then the will be analyzed the differences between each case.

5.1. Prey-Predator model

The prey-predator model, also known as the Lotka-Volterra model, describes the dynamic interaction between two species: one being a predator and the other its prey.

5.1.1. Mathematical formulation

The model is represented by a system of two differential equations that describe the rates of change of the population sizes of the two species over time. Let x represent the prey population and y represent the predator population. Prey and predator population are described by two equations which are respectively:

$$\frac{dx}{dt} = \alpha x - \beta xy - \rho x^2, \tag{5.1}$$

where the derivative of x with respect to t represents the instantaneous growth rate of this population, αx is the natural growth rate of the prey in the absence of predation with α the intrinsic growth rate of the prey, βxy represents the rate of predation upon the prey with β that characterizes the effectiveness of predators at capturing prey. The last term is a sort of "overcrowding coefficient", that is a self-limiting effect on the prey population that prevents it to grow indefinitely.

$$\frac{dy}{dt} = \delta xy - \gamma y, \tag{5.2}$$

where the derivative of y with respect to t represents the instantaneous growth rate of this population, δxy represents the growth of the predator population due to predation and γy is the natural death rate of the predators in the absence of prey.

Thanks to these equations it is possible to understand how these two population would live without each other: In the absence of predators ($y = 0$), the prey population would grow exponentially, but the presence of the self limiting factor ρ avoid this, as indicated by $\frac{dx}{dt} = \alpha x - \rho x^2$ while the predators, in absence of prays ($x = 0$), decrease exponentially, following $\frac{dy}{dt} = -\gamma y$.

The interaction term βxy in the prey equation and δxy in the predator equation demonstrates the nature of prey-predator interaction, indeed the prey population is decreased by predation, while the predator population is supported by the consumption of prey.

5.1.2. Implementation

The complete code for this application is present in (A.1.1). For what concerns numerical values of the parameters just presented in section (5.1) these are all reported in the appendix. The covariance matrix and the measurement noise matrix are taken as the identity matrix of order two, while the process noise is a discrete withe noise (result of a zero mean multi variate normal distribution). The transition function is built with the governing equation of the phenomenon (5.1) and (5.2), while the measurement function indicates the relation between observed data and state variables and in this case the observed variables are exactly the state variables. It is important to clarify that in real applications observed variables are different from the state ones, but the main goal of this section is to validate the correctness of the implemented code.

The exact solution is computed for each time step with equation (5.1) and (5.2) and it is used by the function `get_sensor_reading` as if it were the observed data. At this point the object EnKF and the model can be created through the constructor seen in 3.3. Results are then plotted.

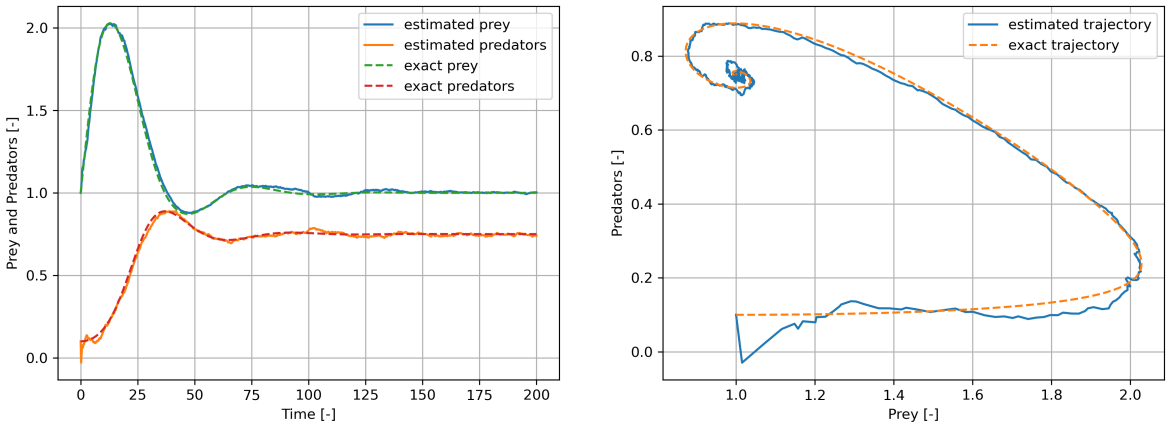


Figure 5.1: Results for classic Prey-Predator

5.2. Prey predator model with state-parameter

There are cases in which the parameters that govern the problems are not known, therefore these parameters become also state variables that need to be found. It is worth noting that these state-parameters are not governed by a differential equations but are taken constant in time. For this reason it have been studied the implementation for the case where the intrinsic growth rate of the prey α is not known.

As for the previous implementations, the complete code for this application is present in (A.1.1) and all the numerical values of the parameters presented in section (5.1) are reported there. The covariance matrix and the measurement noise matrix are taken as the identity matrix of order two, while the process noise is a discrete withe noise (result of a zero mean multi variate normal distribution) and the initial values of the three state variable are assigned by the user.

For what concerns the transition matrix, it is built with the governing equation of the phenomenon (5.1) and (5.2), but it also has the constant one, thanks to which the scalar product $F \cdot x$, where F is the transition function and x is the vector containing the state variable, returns exactly the state-parameter α, which is taken constant in time. The measurement function indicates the relation between observed data and state variables and in this case the observed variables are exactly the first two state variables. The state-parameter α is not taken into consideration here because there is no way of measuring this value. It is important to clarify that in real applications observed variables are different from the state ones but the main goal of this section is to validate the correctness of the implemented code. The exact solution is computed for each time step with equation (5.1)

and (5.2) and it is used by the function `get_sensor_reading` as it were the observed data. At this point the object EnKF and the model can be created through the constructor seen in 3.3. Results are then plotted.

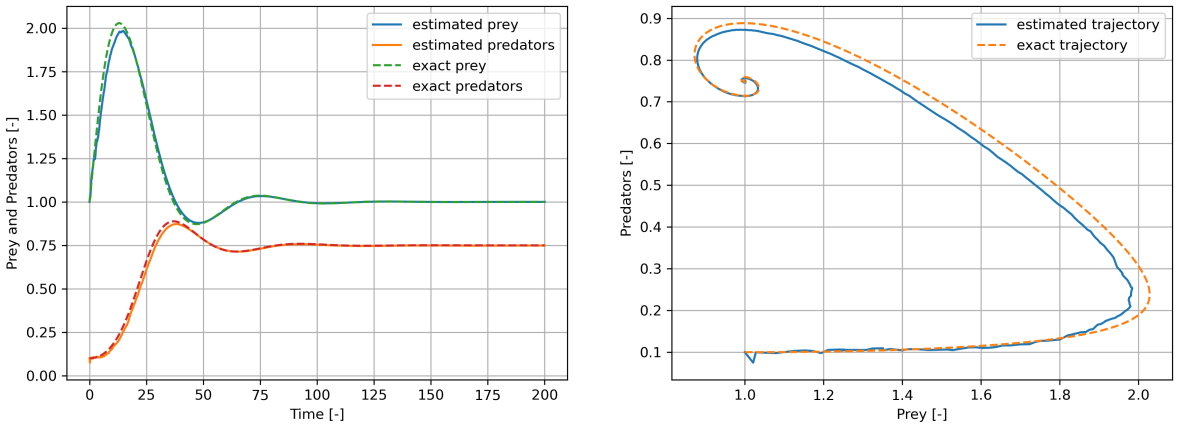
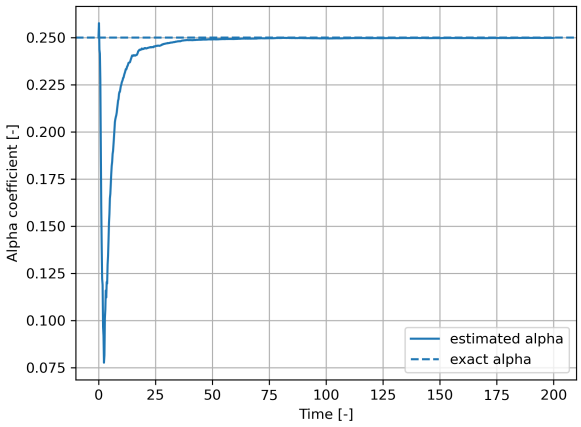


Figure 5.2: Results for Prey-Predator with state-parameter



5.3. Prey-Predator model with a simple surrogate model

As for the previous implementations, the complete code for this application is present in (A.1.1) and all the numerical values of the parameters presented in section (5.1) are reported there. The covariance matrix and the measurement noise matrix are taken as the identity matrix of order two, while the process noise is a discrete withe noise (result of a zero mean multi variate normal distribution) and the initial values of the state variable are assigned by the user in the first part of the code.

For this example, the transition function is built from a neural network that takes as input the array containing state variables at a specific time and returns the state variables at the subsequent time step. The measurement function indicates the relation between observed data and state variables and in this case the observed variables are exactly the first two state variables. It is important to clarify that in real applications observed variables are different from the state ones but the main goal of this section is to validate the correctness of the implemented code. The exact solution is computed for each time step with equation (5.1) and (5.2) and it is used by the function `get_sensor_reading` as it were the observed data. At this point the object EnKF and the model can be created through the constructor seen in 3.3. Results are then plotted.

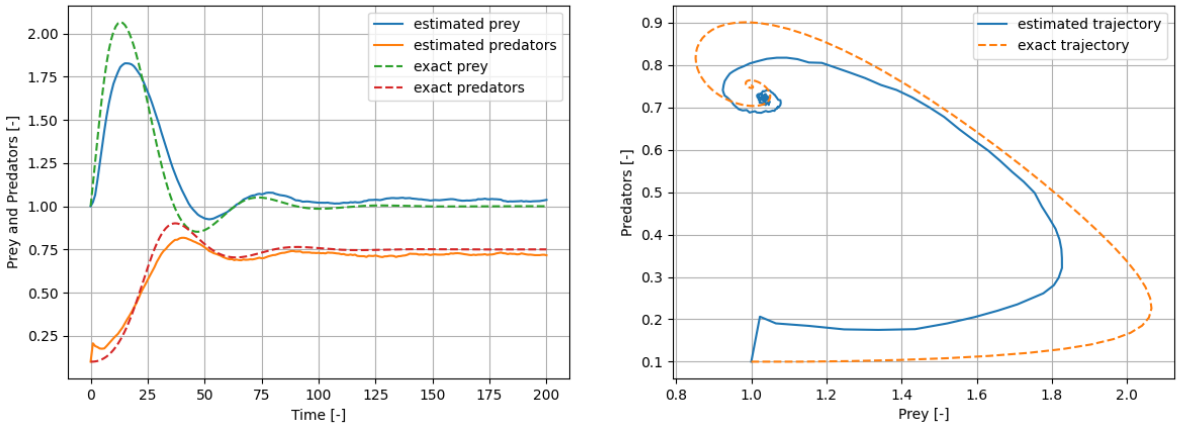


Figure 5.3: Results for Prey-Predator with surrogate model

6 | FNO numerical examples

In this chapter some numerical examples are presented, in order to show some applications of the Fourier Neural Operator classes implemented and depicted in chaoter 4. In particular, the numerical examples here presented deal with the Burgers equation (1D model) and the Darcy flow equation (2D model).

6.1. 1D Burgers equation

Burgers' equation, also known as the Bateman–Burgers equation, stands as a fundamental partial differential equation with characteristics of both convection and diffusion [21]. It finds applications across diverse fields in applied mathematics, including fluid mechanics, nonlinear acoustics, gas dynamics, and traffic flow [22]. Harry Bateman initially introduced this equation in 1915 [3], and its further exploration and study were undertaken by Johannes Martinus Burgers in 1948 [7].

6.1.1. Mathematical formulation

Given a field $u(x,t)$ and a diffusion coefficient $\nu \in \mathbb{R}_+$, the general form of Burgers' equation (also known as viscous Burgers' equation) in one space dimension is:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{6.1}$$

Equation 6.1 is referred as the *advective form* of Burgers equation. An alternative version, more suitable for numerical integration, is the *conservative form* in equation 6.2.

$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u^2}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{6.2}$$

In this application, the domain considered is $\Omega = (x,t) \in (0,1) \times (0,1]$, the viscosity coefficient is $\nu = 0.1$ and the problem is formulated as:

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u^2}{\partial x} &= \nu \frac{\partial^2 u}{\partial x^2} \qquad (x,t) \in \Omega \\ u(x,0) &= u_0 \qquad x \in (0,1) \end{aligned} \tag{6.3}$$

with $u_0 \in L^2((0,1);\mathbb{R})$.

The aim of this example is to learn the map between the initial condition u_0 to the solution at time $t = 1$, namely:

$$g^\dagger : L^2((0,1);\mathbb{R}) \to H^r((0,1);\mathbb{R}) \qquad u_0(x) \mapsto u(x,1) \qquad \forall r > 0$$

6.1.2. Training data

In order to train a FNO for learning the map $G^\dagger$, it is necessary to pick some $u_0 \in L^2((0,1);\mathbb{R})$ and to develop a method for computing the corresponding *exact* solution at each time t, for then extracting the solution at the end time.

Computing reference solution The reference exact solution $u(x,t)$ is approximated by means of a finite difference scheme, so that the corresponding discretized solution u_h is:

$$\begin{aligned} u_h(x_i, t_{j+1}) &= u_h(x_i, t_j) + \\ &+ dt\left[\nu \frac{u_h(x_{i+1}, t_j) - 2u_h(x_i, t_j) + u_h(x_{i-1}, t_j)}{dx^2} - \frac{u_h(x_i, t_j)^2 - u_h(x_{i-1}, t_j)^2}{2dx}\right] \end{aligned} \tag{6.4}$$

where x_i and t_j belongs to appropriate partitions of the space and time domain. In order to achieve accurate results, a fine grid in time and space is required. In this case 2^6 points in space and 2^{17} points in time are considered.

Initial condition In order to consider general initial conditions in the training data, a random normal distribution of values $u_{0k} = u_0(x_k)$ with zero mean and unitary variance is created. The partition considered for choosing the points x_k is less fine then the partition considered for the finite difference scheme. Indeed, if the same partition was used, the resulting initial condition would be merely noise. In this example the initial random values are taken on a partition four times less fine then the partition considered in the finite difference scheme and it is then interpolated on the finer partition by a third degree piecewise polynomial interpolation.

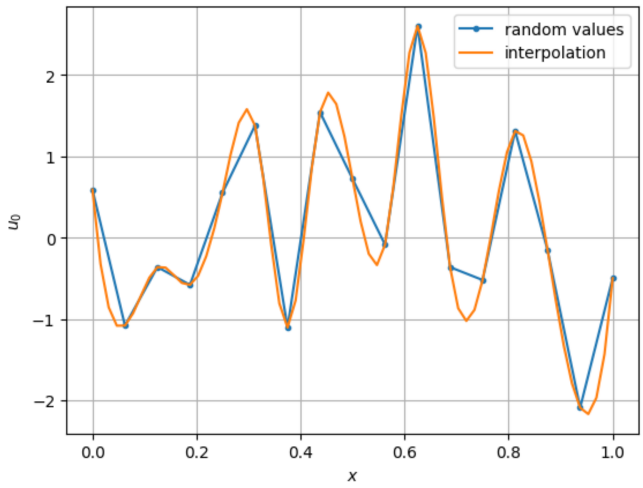


Figure 6.1: Example of construction of the initial condition by interpolating random values on a coarser grid

The training data is made of 800 pairs (u_0, u_f), with $u_f(x) = u(x, 1)$, divided in 640 pairs for the actual training and 160 pairs for validation. A complete python code for generating these data is presented in B.1.1

6.1.3. Creating the FNO model

As a first step for setting an FNO model, some useful functions for controlling the learning process have been created. In general, the following strategies has been used:

- An inverse time decay learning step has been adopted, in order to reduce the learning rate as the process goes on.
- The optimizer chosen is the standard `keras.optimizer.Adam`.
- An early stopping criterion based on the validation dataset has been set, for stopping the process in case of excessive overfitting.
- The loss considered is the mean squared error. Also other metrics have been monitored during the learning phase.
- Both dropout and kernel regularization has been exploited, for reducing overfitting.

The actual FNO model considered is made of 11 Fourier-Layers with 7 modes each, while the P-Layer is a dense layer with 256 neurons. The complete code for implementing such a model is reported in B.1.2.

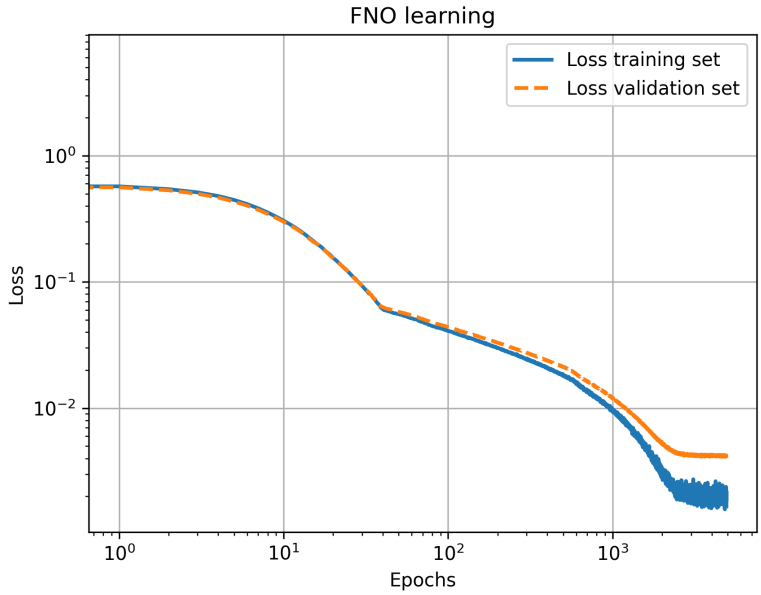
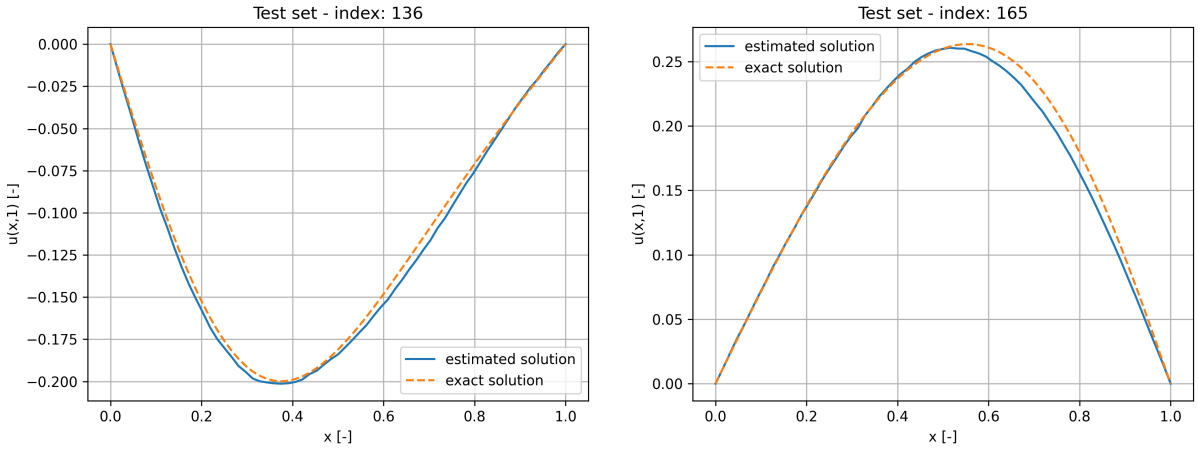
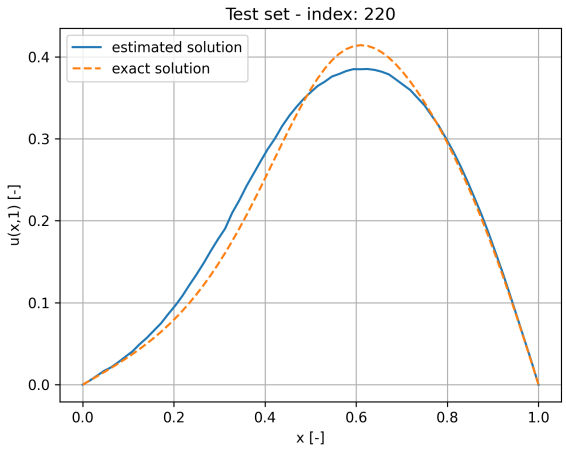


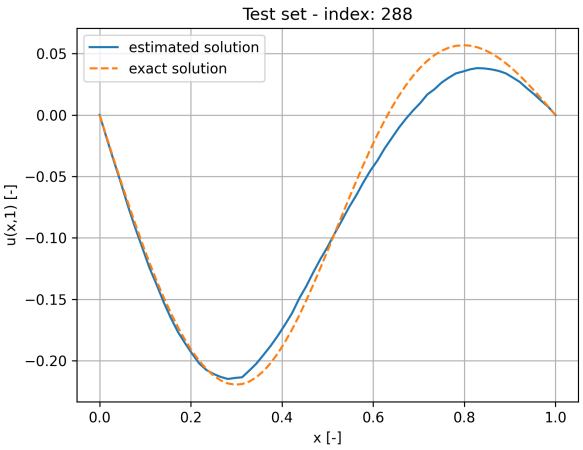
Figure 6.2: Learning path of the FNO model for Burgers equation

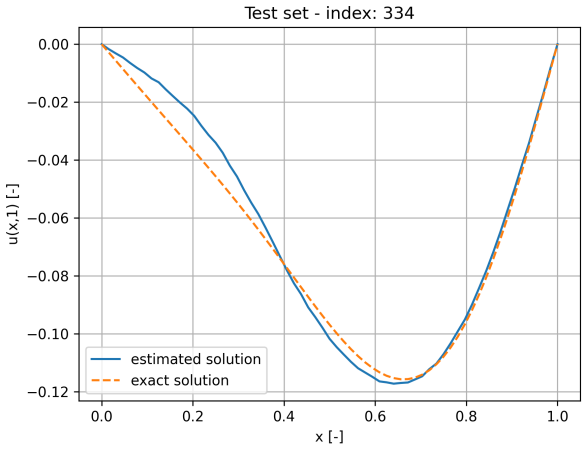
6.1.4. Results

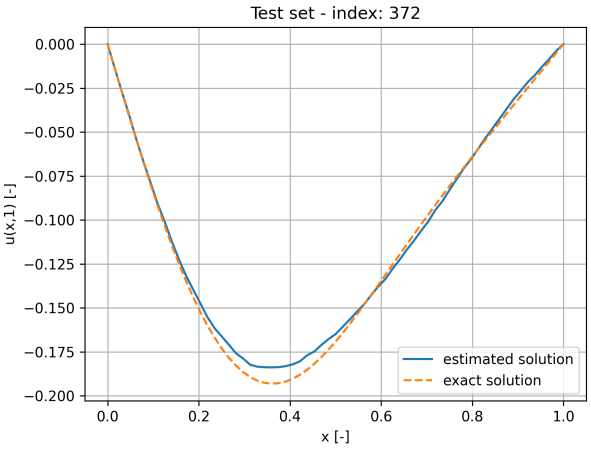
In the following some random results computed on the validation set is shown. Remind that these data have not been used during the training of the model. The code for reproducing such results is shown in B.1.3.

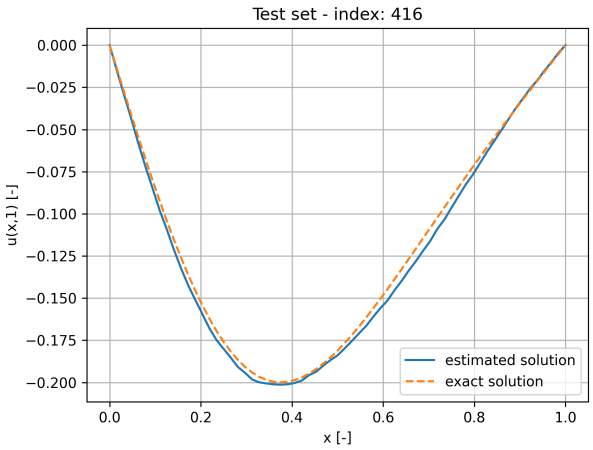


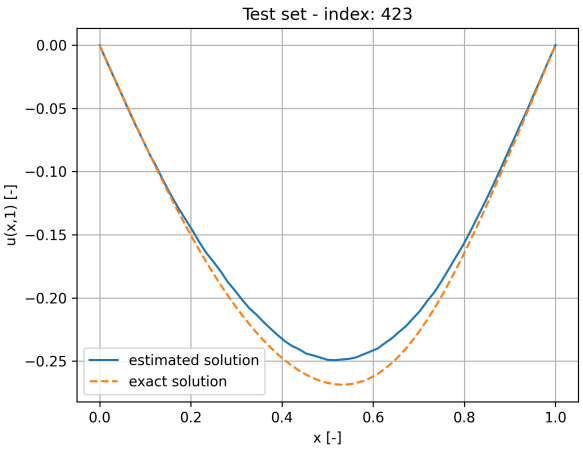

Test set - index: 220
estimated solution
exact solution
u(x,1) [-]
x [-]

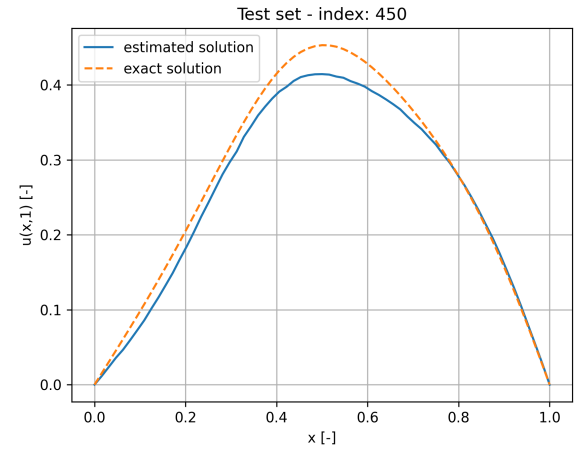

Test set - index: 288
estimated solution
exact solution
u(x,1) [-]
x [-]

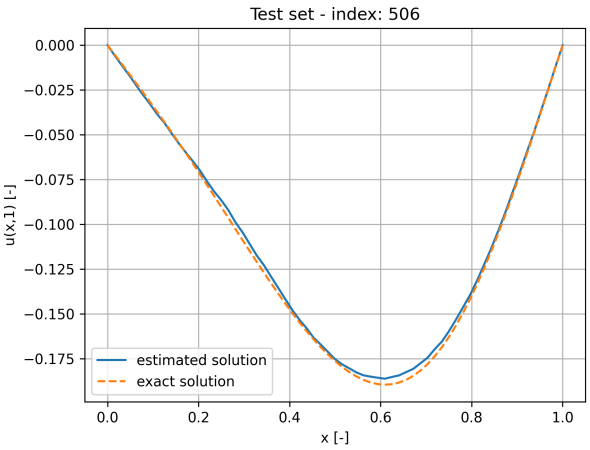

Test set - index: 334
estimated solution
exact solution
u(x,1) [-]
x [-]

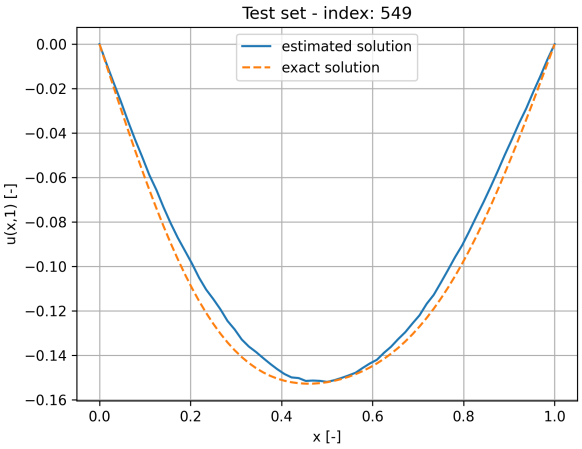

Test set - index: 372
estimated solution
exact solution
u(x,1) [-]
x [-]

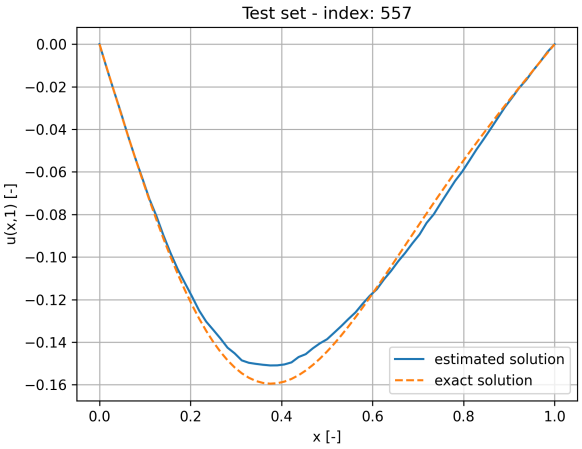

Test set - index: 416
estimated solution
exact solution
u(x,1) [-]
x [-]

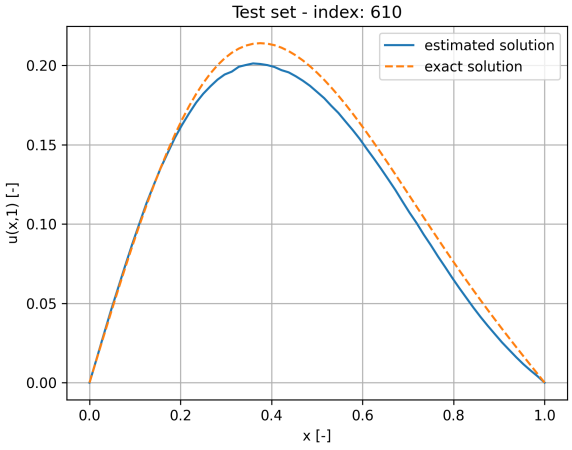

Test set - index: 423
estimated solution
exact solution
u(x,1) [-]
x [-]

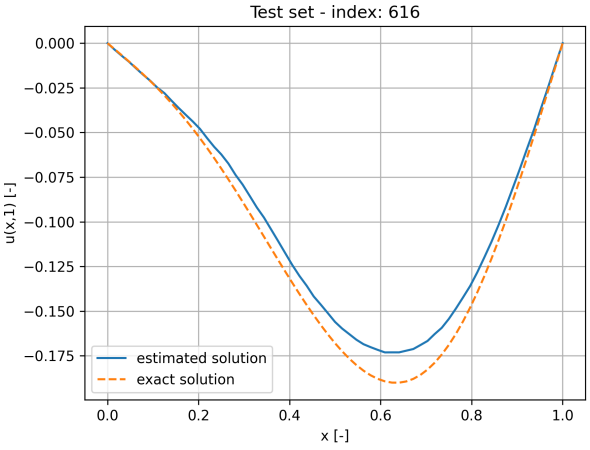

Test set - index: 450
estimated solution
exact solution
u(x,1) [-]
x [-]


Test set - index: 506
estimated solution
exact solution
u(x,1) [-]
x [-]


Test set - index: 549
estimated solution
exact solution
u(x,1) [-]
x [-]


Test set - index: 557
estimated solution
exact solution
u(x,1) [-]
x [-]


Test set - index: 610
estimated solution
exact solution
u(x,1) [-]
x [-]


Test set - index: 616
estimated solution
exact solution
u(x,1) [-]
x [-]

6.2. 2D Darcy flow equation

Darcy flow equation is a second order linear elliptic PDE that rules several physical phenomena including the pressure of subsurface flow, the deformation of linearly elastic materials, and the electric potential in conductive mater. In this example, the 2D steady-state equation is considered.

6.2.1. Mathematical formulation

Consider the set $\Omega = (0,1)^2$, given a spatially varying diffusion coefficient $a \in L^\infty(\Omega;\mathbb{R})$ and a forcing term $f \in L^2(\Omega;\mathbb{R})$ the Darcy equation describing the two-dimensional flow $u(x,y)$ with homogeneous Dirichlet boundary conditions reads:

$$\begin{aligned} -\text{div}(a\nabla u) &= f \qquad (x,y) \in \Omega \\ u &= 0 \qquad (x,y) \in \partial\Omega \end{aligned} \tag{6.5}$$

In this example the goal is to learn the operator mapping the diffusion coefficient to the solution, namely:

$$g^\dagger : L^\infty(\Omega;\mathbb{R}) \to H_0^1(\Omega;\mathbb{R}) \qquad a(x,y) \mapsto u(x,y)$$

6.2.2. Training Data

The training dataset considered is taken from the same example present in [18], in particular all the data are available at `https://github.com/NeuralOperator/neuraloperator`. Once the data has been downloaded, some little manipulation was carried out, in order to make them fit for the `keras` model. The complete code for the data manipulation is reported in B.2.1

6.2.3. Creating the FNO model

Also in this case, as a first step for setting an FNO model, some useful functions for controlling the learning process have been created. In general, the following strategies has been used:

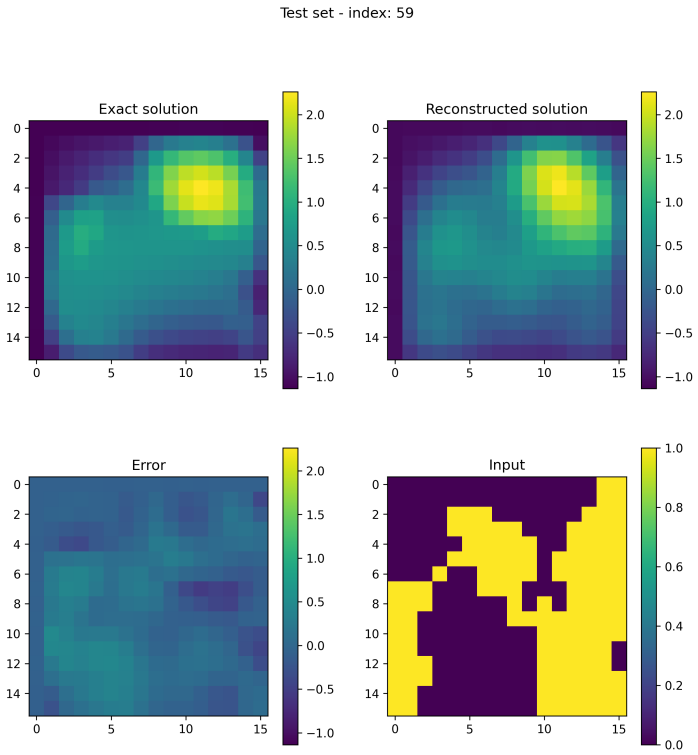
- An inverse time decay learning step has been adopted, in order to reduce the learning rate as the process goes on.
- The optimizer chosen is the standard `keras.optimizer.Adam`.

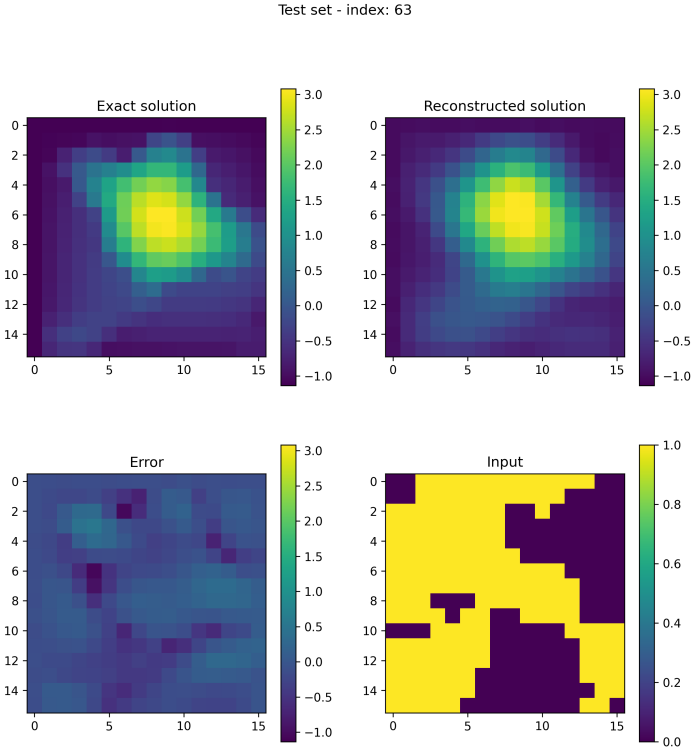
- An early stopping criterion based on the validation dataset has been set, for stopping the process in case of excessive overfitting.
- The loss considered is the mean squared error. Also other metrics have been monitored during the learning phase.
- Both dropout and kernel regularization has been exploited, for reducing overfitting.

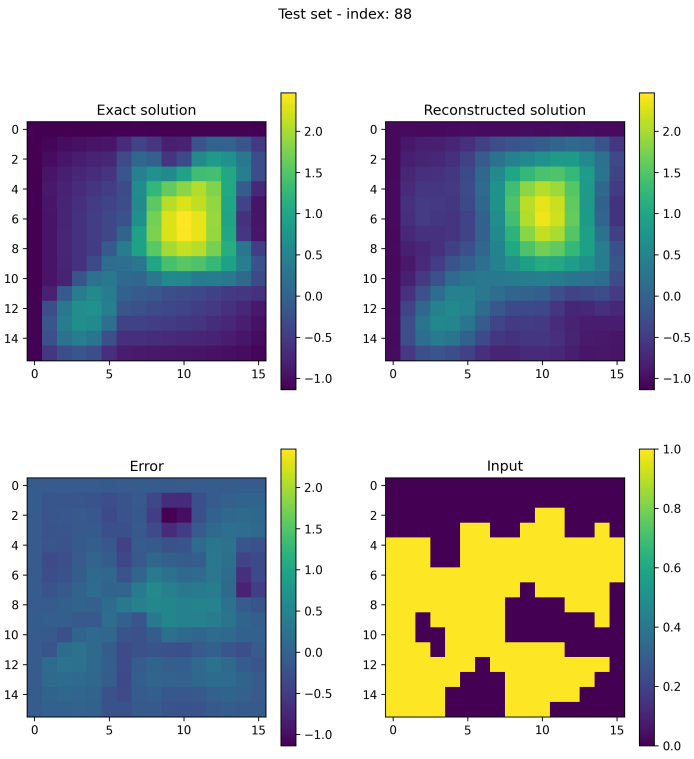
The actual FNO model considered is made of 11 Fourier-Layers with 7 modes each, while the P-Layer is a dense layer with 32^2 neurons. The complete code for implementing such a model is reported in B.2.2.

6.2.4. Results

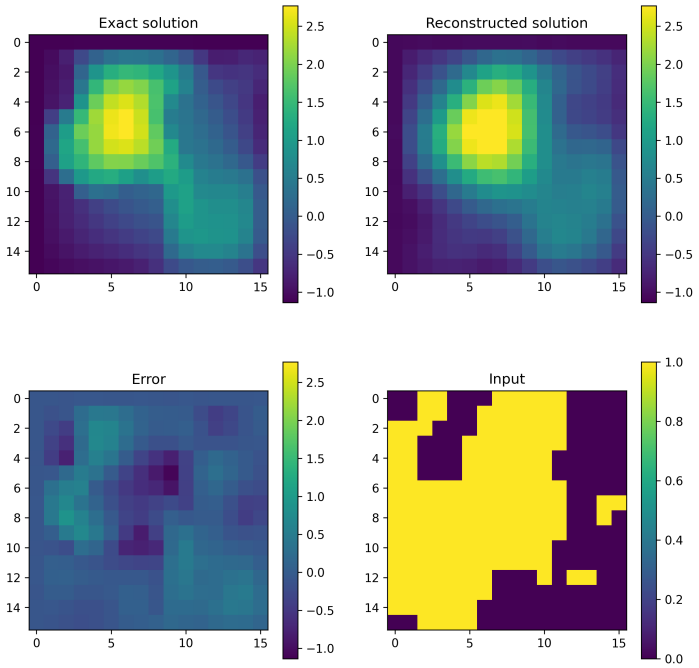
In the following some random results computed on the validation set is shown. Remind that these data have not been used during the training of the model. The code for reproducing such results is shown in B.2.3.



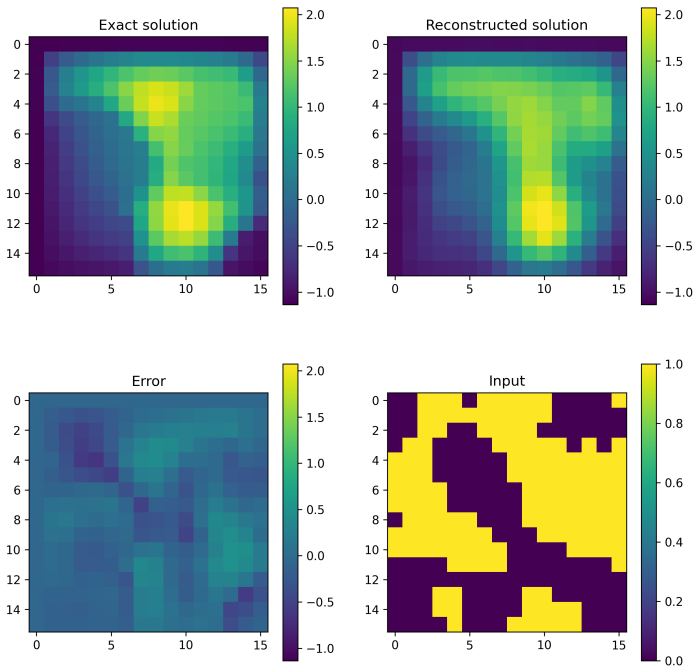

Test set - index: 63
Exact solution
Reconstructed solution
Error
Input


Test set - index: 88
Exact solution
Reconstructed solution
Error
Input

Test set - index: 109



Test set - index: 161



7 | A coupled EnKF - FNO model

7.1. Predicting Burgers Equation in time

The first application consisting in a combined model adopting both the already discussed EnFK model and the Fourier Neural operator is a model for predicting the variation in time of the solution for the 1D Burgers equation discussed in section 6.1. The time advancing and predictions are carried out by exploiting an EnKF model in which the transition function is represented by the FNO model, similarly as what discussed in 5.3.

7.1.1. Constructing the FNO model for time advancing

Recall the Burgers equation in its advective form, as introduced in section 6.1:

$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u^2}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \tag{7.1}$$

In this application, the domain considered is $\Omega = (x,t) \in (0,1) \times (0,0.25]$, the viscosity coefficient is $\nu = 0.025$ and the problem is formulated as:

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u^2}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \qquad & (x,t) \in \Omega \\ u(x,0) = u_0 \qquad & x \in (0,1) \end{aligned} \tag{7.2}$$

with $u_0 \in L^2((0,1);\mathbb{R})$.

The aim of this FNO is to learn the map from the solution at a time t to the solution at time $t+\tau$, namely:

$$g^\dagger : H^r((0,1);\mathbb{R}) \to H^r((0,1);\mathbb{R}) \qquad u(x,t) \mapsto u(x,t+\tau) \qquad \forall r > 0$$

The strategy adopted for computing the reference solution and the initial condition are exactly the same as described in 6.1.2. The dataset generated consist of 3600 training samples and 900 test samples. The time interval between the input and the output of the

model has been selected to be $\tau = 1/64$. A complete python code for generating these data is presented in C.1.1.

For creating the FNO model, the following strategies has been used:

- An inverse time decay learning step has been adopted, in order to reduce the learning rate as the process goes on.
- The optimizer chosen is the standard `keras.optimizer.Adam`.
- An early stopping criterion based on the validation dataset has been set, for stopping the process in case of excessive overfitting.
- The loss considered is a custom loss consisting in the sum between the cosine similarity loss and the Huber loss multiplied by a factor of 30.
- Both dropout and kernel regularization has been exploited, for reducing overfitting.
- Each input and output sample has been normalized by a factor equal to the maximum of the absolute value of the input, for the sake of generalization.

The actual FNO model considered is made of 3 Fourier-Layers with 14 modes each, while the P-Layer is a dense layer with 512 neurons. The complete code for implementing such a model is reported in C.1.2.

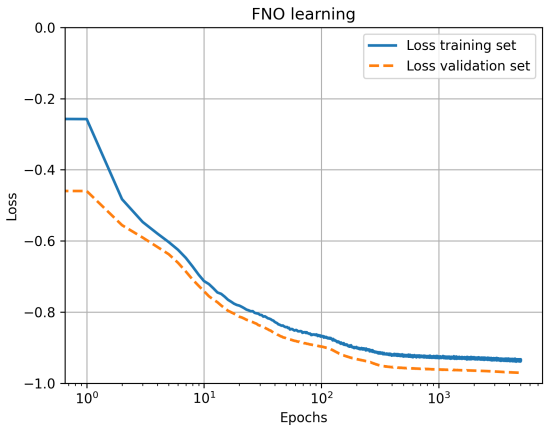
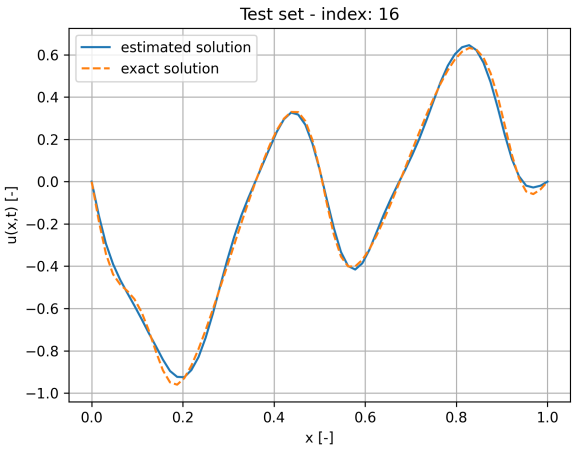
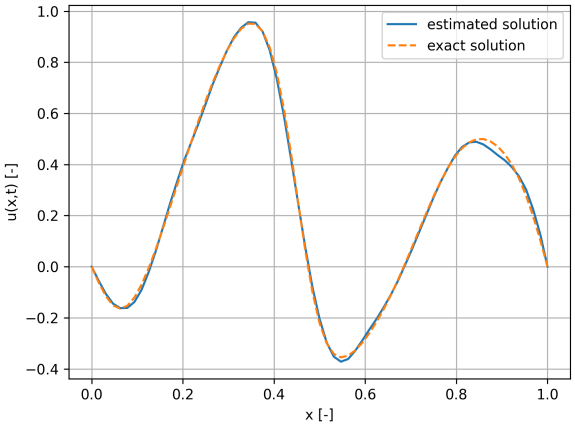
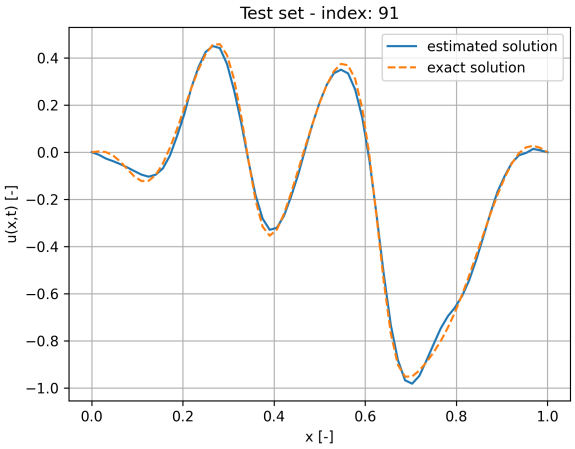


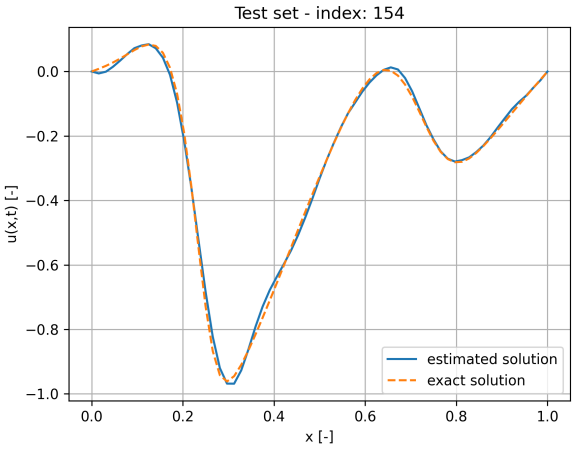
Figure 7.1: Learning path of the FNO model for Burgers equation in time. Note that cosine similarity loss ranges from -1 to 1, with -1 being the best similarity score.

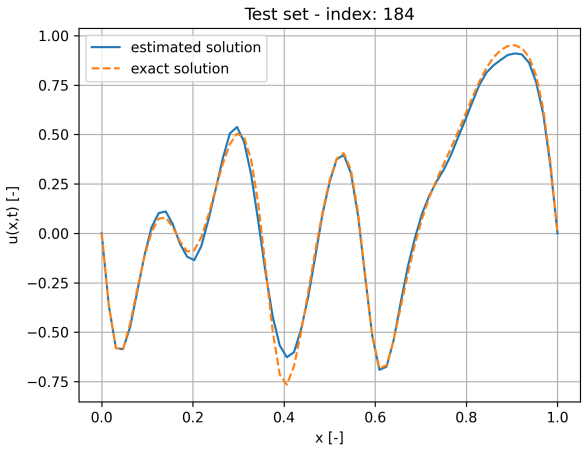
In the following some random results computed on the validation set is shown. Remind that these data have not been used during the training of the model. The code for reproducing such results is shown in C.1.3.

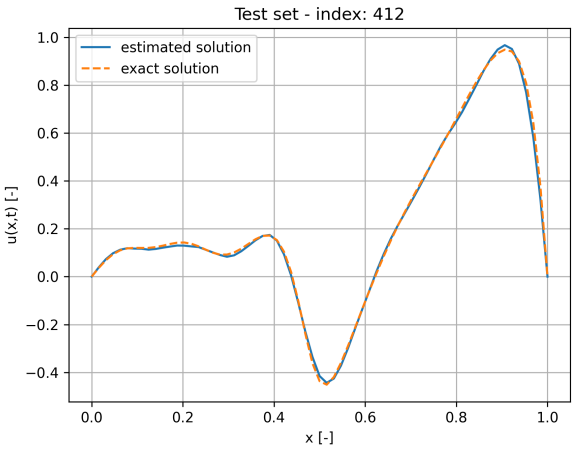

Test set - index: 16
estimated solution
exact solution
u(x,t) [-]
x [-]

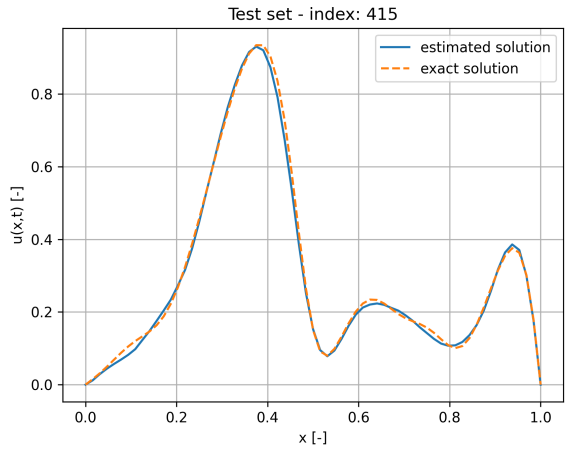

Test set - index: 60
estimated solution
exact solution
u(x,t) [-]
x [-]

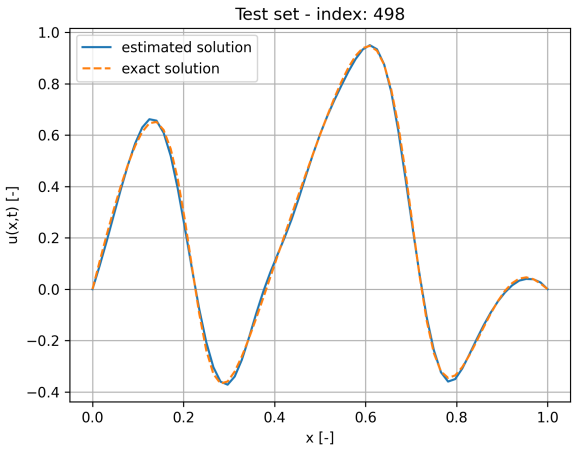

Test set - index: 91
estimated solution
exact solution
u(x,t) [-]
x [-]

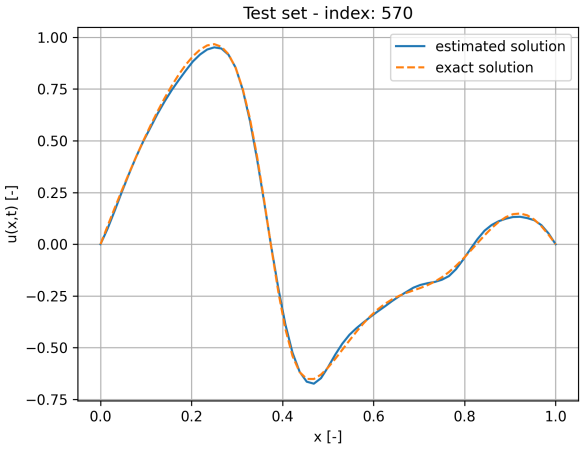

Test set - index: 154
estimated solution
exact solution
u(x,t) [-]
x [-]

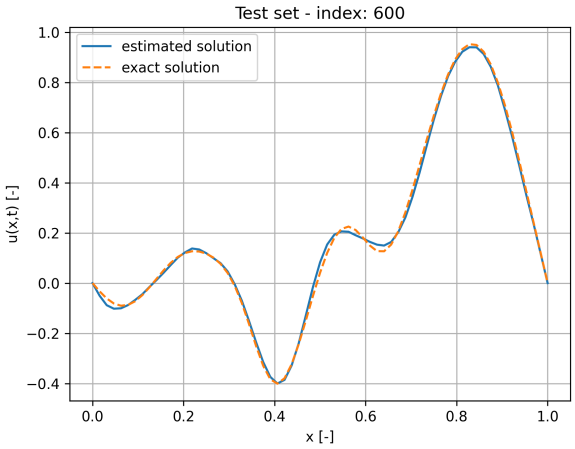

Test set - index: 184
estimated solution
exact solution
u(x,t) [-]
x [-]

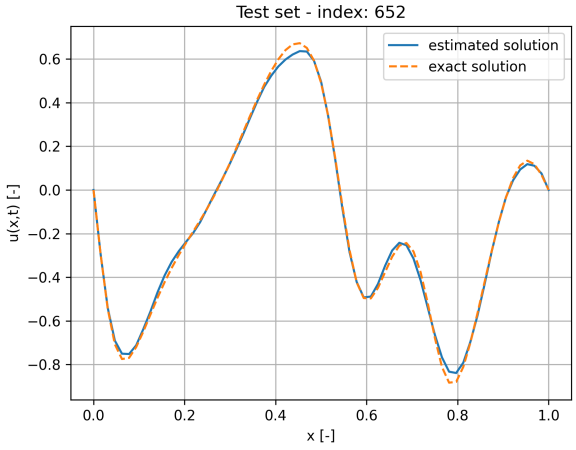

Test set - index: 412
estimated solution
exact solution
u(x,t) [-]
x [-]

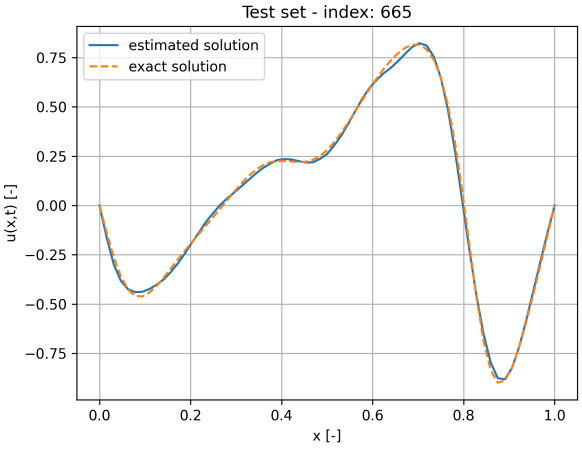

Test set - index: 415
estimated solution
exact solution
u(x,t) [-]
x [-]


Test set - index: 498
estimated solution
exact solution
u(x,t) [-]
x [-]


Test set - index: 570
estimated solution
exact solution
u(x,t) [-]
x [-]


Test set - index: 600
estimated solution
exact solution
u(x,t) [-]
x [-]


Test set - index: 652
estimated solution
exact solution
u(x,t) [-]
x [-]


Test set - index: 665
estimated solution
exact solution
u(x,t) [-]
x [-]

7.1.2. Setting the EnKF model

Once the Fourier Neural Operator is trained, it can be used as a transition function in an EnKF model. Since the FNO was trained on normalized data, an appropriate scaling of the input and output of the transition function is needed.

In order to be consistent with the FNO, the time interval used in the EnKF model has to be the same time interval τ used to create training data for the Neural Operator. Instead, the total width of the time window over which the simulation is carried out could be longer, in particular a final time of 0.5 has been selected. The spatial discretization of the function u makes the dimension of the problem to be 65, meaning that for each time step, the model would make a prediction on the 65 state variable that are the spatial components of the solution u.

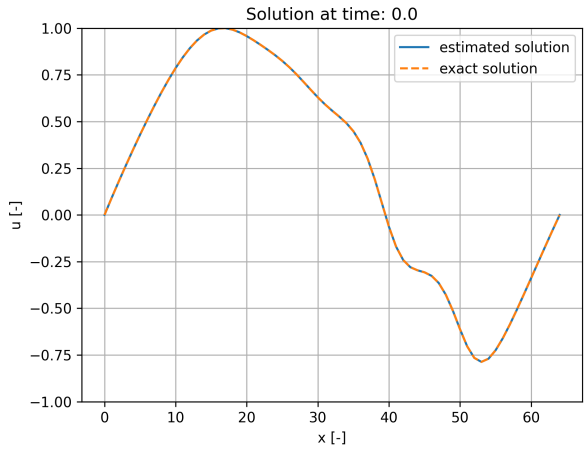
In this application, the measurement function provides the reference exact solution computed as in section 6.1.2 for the specific time step in which it is invoked. Obviously, in a practical case in which an exact solution is not known, the measurement function will provide some experimental data that can be linked to the state variables. So, in this case the dimension of the measurements is the same as the state variables.

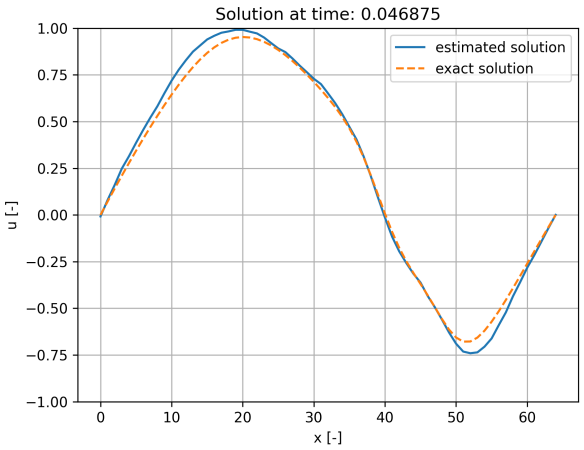
The covariance matrix has been set to be the actual covariance of the exact solution. This is just in order to show the correctness of this method, in a real scenario the covariance matrix can be deduced by a-priory knowledge on the phenomenon or by empirical testing. The measurement noise matrix and the process noise matrix have been set to be respectively $2\mathbf{I}$ and $0.01\mathbf{I}$, where $\mathbf{I}$ is the identity matrix of dimension 65.

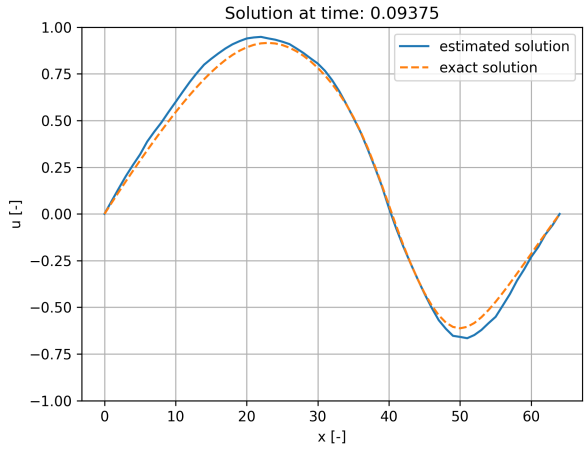
The number of ensemble generated by the EnKF model has been set to 10000, which led to quite heavy computational costs, but this is necessary for having a good prediction on the subsequent time steps, given the various uncertainties introduced in the model (especially by the FNO model). A complete python code for reproducing this setting is presented in C.1.4.

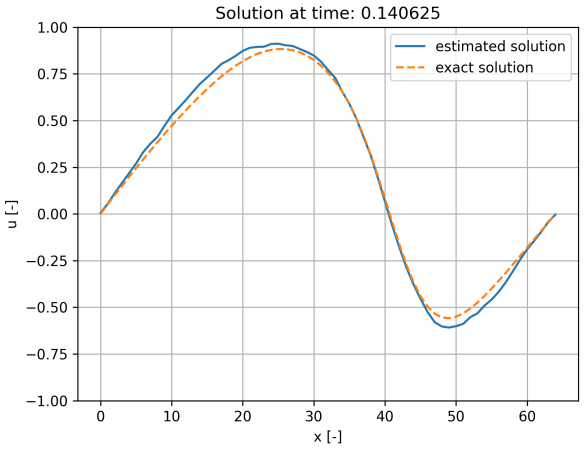
7.1.3. Results

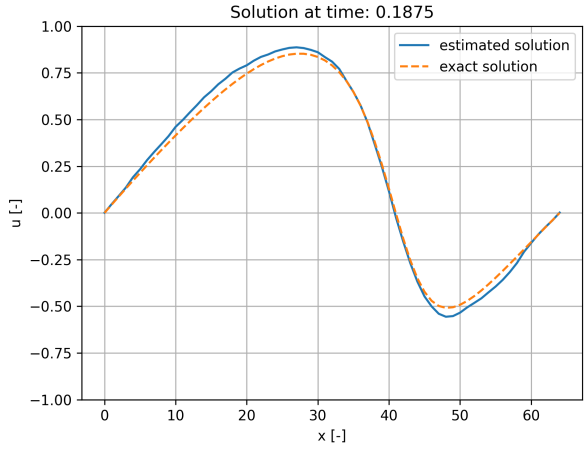
Results consist in the subsequent predictions of the solution for each time step. In the following just some of the time steps computed have been reported.

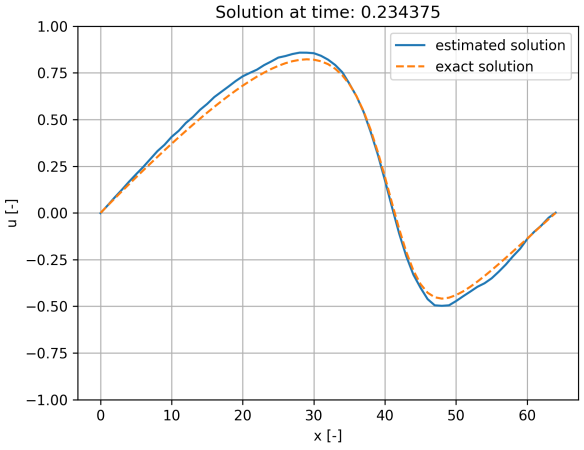

Solution at time: 0.0
estimated solution
exact solution
u [-]
x [-]

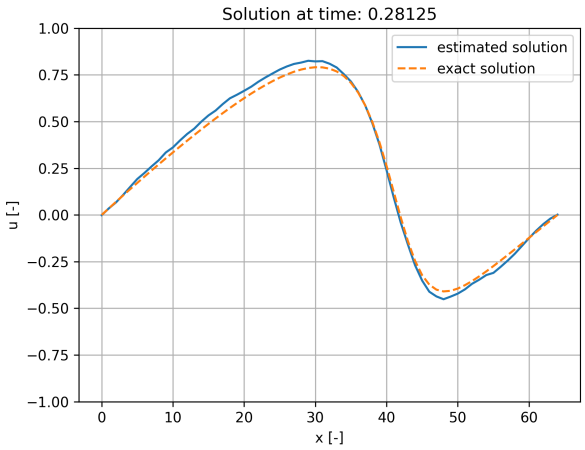

Solution at time: 0.046875
estimated solution
exact solution
u [-]
x [-]

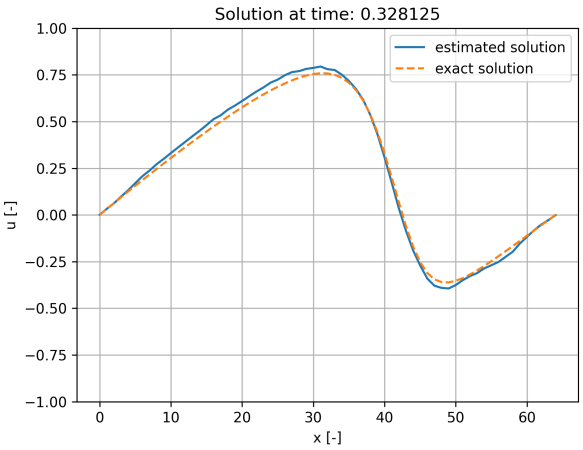

Solution at time: 0.09375
estimated solution
exact solution
u [-]
x [-]

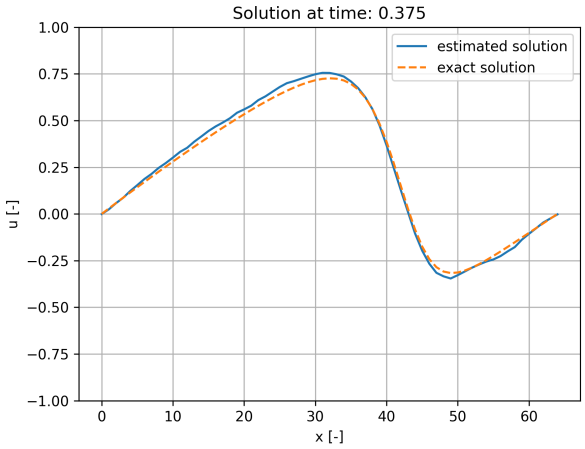

Solution at time: 0.140625
estimated solution
exact solution
u [-]
x [-]

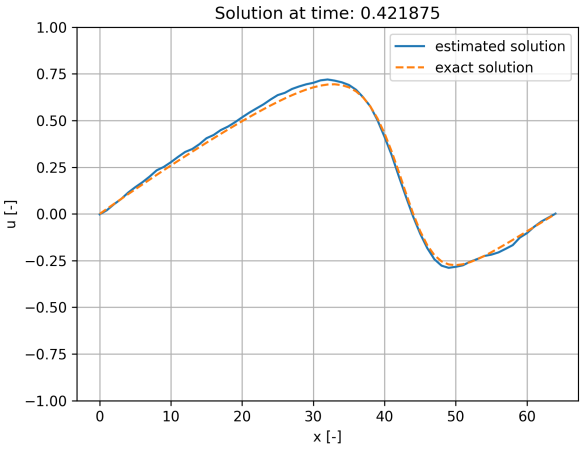

Solution at time: 0.1875
estimated solution
exact solution
u [-]
x [-]

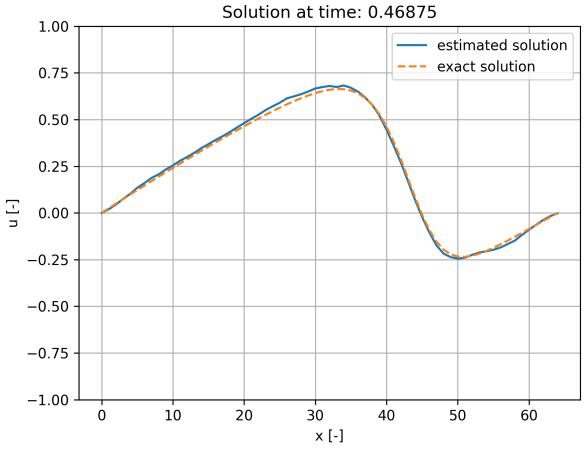

Solution at time: 0.234375
estimated solution
exact solution
u [-]
x [-]

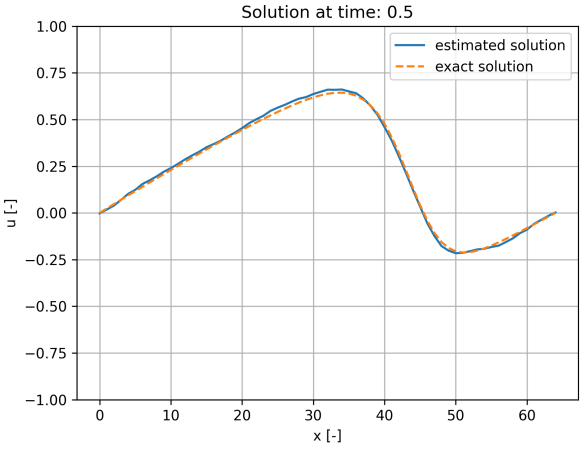

Solution at time: 0.28125
estimated solution
exact solution
u [-]
x [-]


Solution at time: 0.328125
estimated solution
exact solution
u [-]
x [-]


Solution at time: 0.375
estimated solution
exact solution
u [-]
x [-]


Solution at time: 0.421875
estimated solution
exact solution
u [-]
x [-]


Solution at time: 0.46875
estimated solution
exact solution
u [-]
x [-]


Solution at time: 0.5
estimated solution
exact solution
u [-]
x [-]

7.2. Evolving Burgers equation with state-parameter

This application of the EnKF model combined with the Fourier Neural Operator deals with the time evolution of the 1D Burgers equation where the non-linear term is multiplied by a parameter, here denoted as c. This particular case combine all the features seen in section 7.1 with the additional presence of the state-parameter c, so that the EnKF model will have one more unknown, similarly to what seen in 5.2.

7.2.1. Constructing the FNO model for time advancing

In this case, the Burgers equation in its advective form reads:

$$\frac{\partial u}{\partial t} + \frac{1}{2}c\frac{\partial u^2}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{7.3}$$

In this application, the domain considered is $\Omega = (x,t) \in (0,1) \times (0,0.25]$, the viscosity coefficient is $\nu = 0.025$ and the problem is formulated as:

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{1}{2}c\frac{\partial u^2}{\partial x} &= \nu\frac{\partial^2 u}{\partial x^2} \qquad && (x,t) \in \Omega \\ u(x,0) &= u_0 && x \in (0,1) \end{aligned} \tag{7.4}$$

with $u_0 \in L^2((0,1);\mathbb{R})$ and $c \in \mathbb{R}$.

The aim of this FNO is to learn the map from the solution at a time t to the solution at time $t+\tau$, given the state-parameter c namely:

$$g^\dagger : H^r((0,1);\mathbb{R}) \times \mathbb{R} \to H^r((0,1);\mathbb{R}) \qquad (u(x,t),c) \mapsto u(x,t+\tau) \qquad \forall r > 0$$

The strategy adopted for computing the reference solution and the initial condition are exactly the same as described in 6.1.2. The parameter c is generated as a random value in the interval $[0.5, 1.5]$. The dataset generated consist of 4408 training samples and 1472 test samples. The time interval between the input and the output of the model has been selected to be $\tau = 1/64$. A complete python code for generating these data is presented in C.2.1.

For creating the FNO model, the following strategies has been used:

- An inverse time decay learning step has been adopted, in order to reduce the learning rate as the process goes on.
- The optimizer chosen is the standard `keras.optimizer.Adam`.

- An early stopping criterion based on the validation dataset has been set, for stopping the process in case of excessive overfitting.
- The loss considered is a custom loss consisting in the sum between the cosine similarity loss and the Huber loss multiplied by a factor of 10.
- Both dropout and kernel regularization has been exploited, for reducing overfitting.
- Each input and output sample has been normalized by a factor equal to the maximum of the absolute value of the input, for the sake of generalization.

The actual FNO model considered is made of 3 Fourier-Layers with 14 modes each, while the P-Layer is a dense layer with 512 neurons. The complete code for implementing such a model is reported in C.2.2.

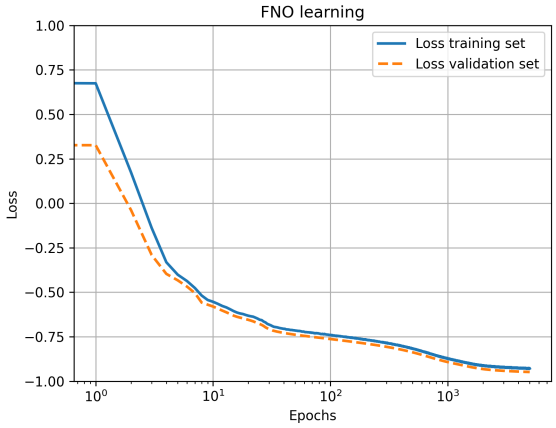


Figure 7.2: Learning path of the FNO model for Burgers equation in time with state-parameter. Note that cosine similarity loss ranges from -1 to 1, with -1 being the best similarity score.

7.2.2. Setting the EnKF model

The implementation for an EnKF model for the Burgers' Equation with state-parameter is analogous to what depicted in 7.1.2, with small variations due to the different dimension of the input, now being integrated with the state-parameter.

The state parameter is represented by a vector of the same length of the solution u with respect to time and it has been appended to the latter so that the overall dimension of the input is doubled. This choice has been made for the model to be expandable to situation in which the state-parameter is a function of the space coordinate.

In order to be consistent with the FNO, the time interval used in the EnKF model has to be the same time interval τ used to create training data for the Neural Operator. Instead, the total width of the time window over which the simulation is carried has been set to 0.5. The spatial discretization of the function u and the addition of the state-parameter makes the dimension of the problem to be 130, meaning that for each time step, the model would make a prediction on the 130 state variable that are the spatial components of the solution (u, c).

In this application, the measurement function provides the reference exact solution for u computed as in section 6.1.2 for the specific time step in which it is invoked, with the appended vector representing c. Obviously, in a practical case in which an exact solution is not known, the measurement function will provide some experimental data that can be linked to the state variables. So, in this case the dimension of the measurements is the same as the state variables.

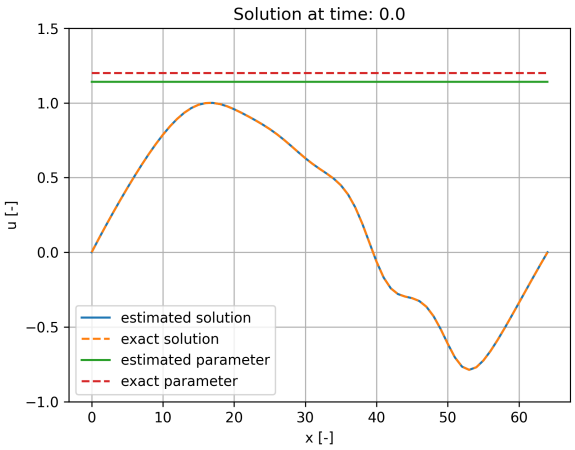
The covariance matrix has been set to be the actual covariance of the exact solution as done in 7.1.2. Also the measurement noise matrix and the process noise matrix have been kept the same as in 7.1.2.

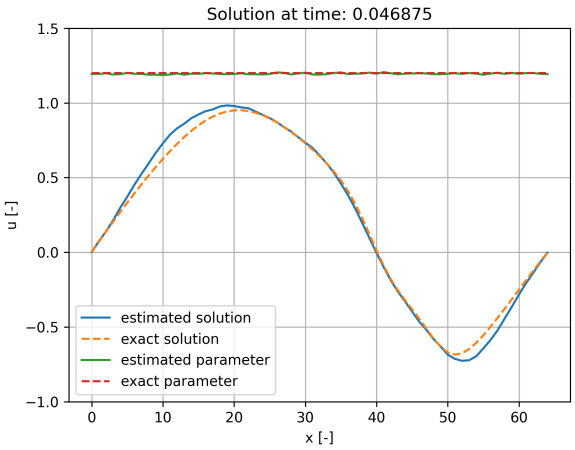
The number of ensemble generated by the EnKF model has been set to 10000, which led to quite heavy computational costs, but this is necessary for having a good prediction on the subsequent time steps, given the various uncertainties introduced in the model (especially by the FNO model). A complete python code for reproducing this setting is presented in C.2.3.

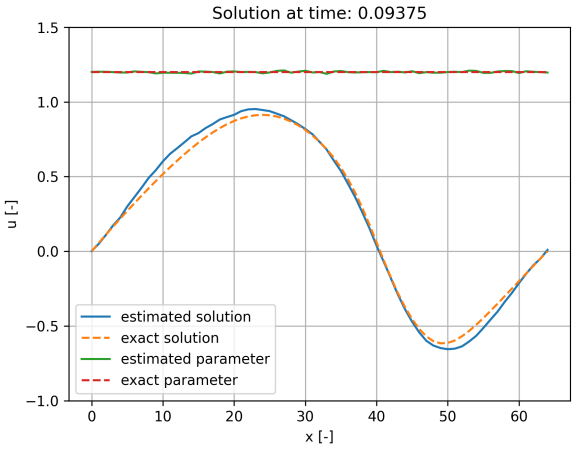
The initial value assigned to the state variable has a small error on the value of the state-parameter, this is to demonstrate the ability of the model to correct this value, provided that a sufficiently good measurements are available.

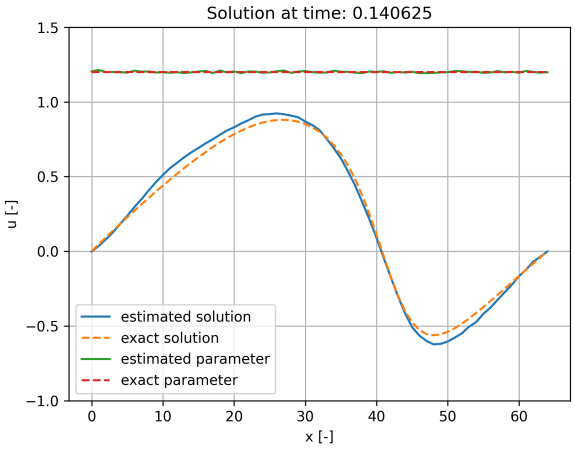
7.2.3. Results

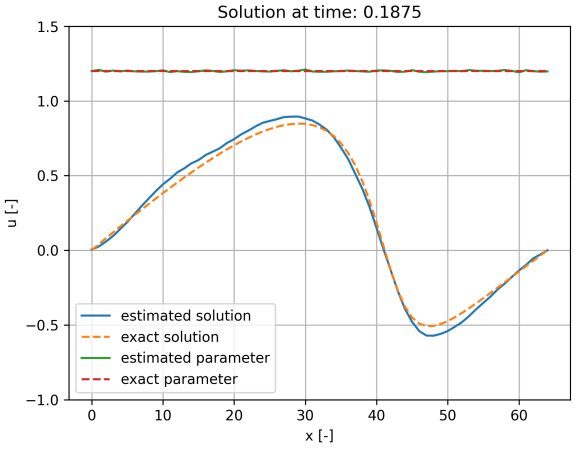
Results consist in the subsequent predictions of the solution for each time step. In the following just some of the time steps computed have been reported.

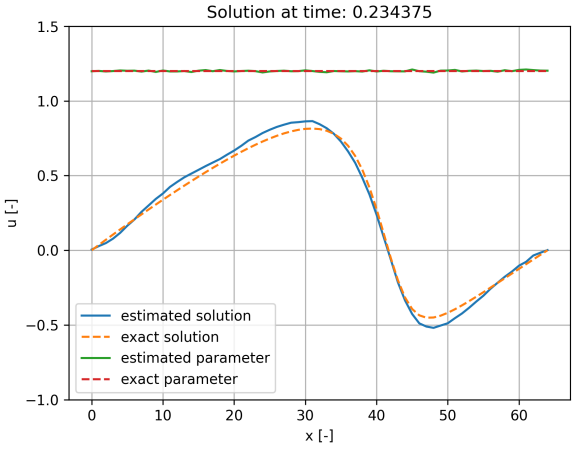

Solution at time: 0.0
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter

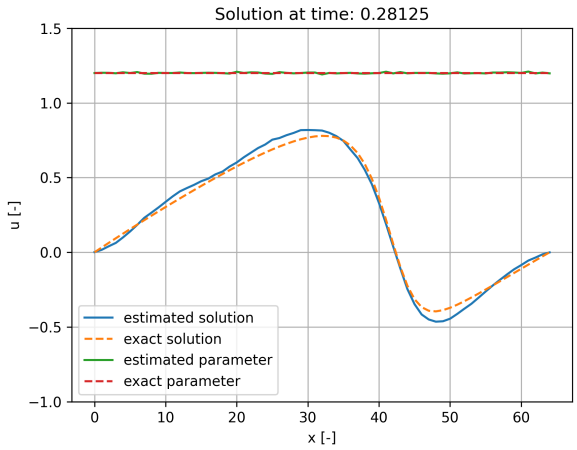

Solution at time: 0.046875
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter

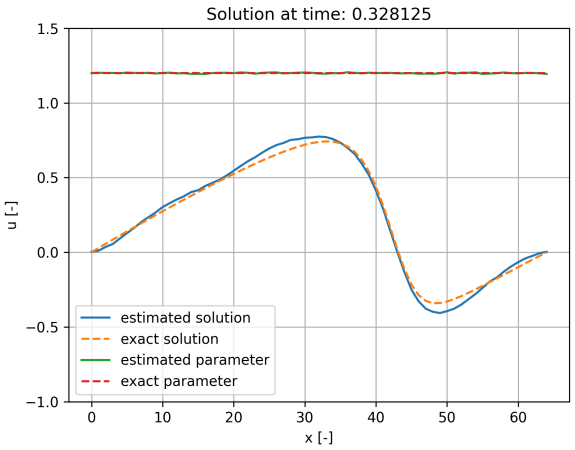

Solution at time: 0.09375
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter

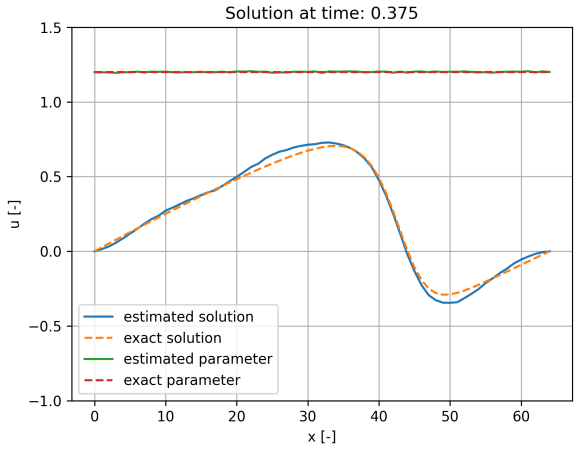

Solution at time: 0.140625
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter

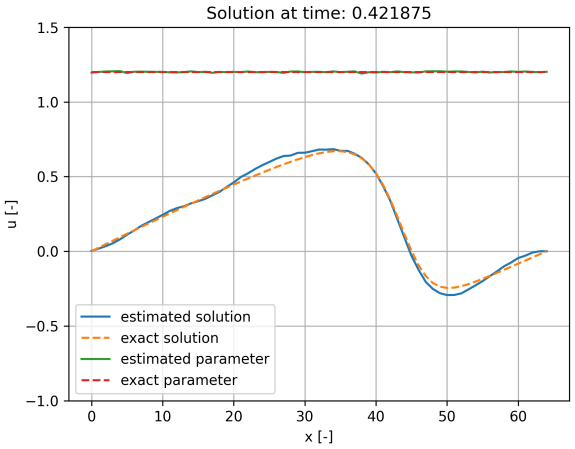

Solution at time: 0.1875
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter

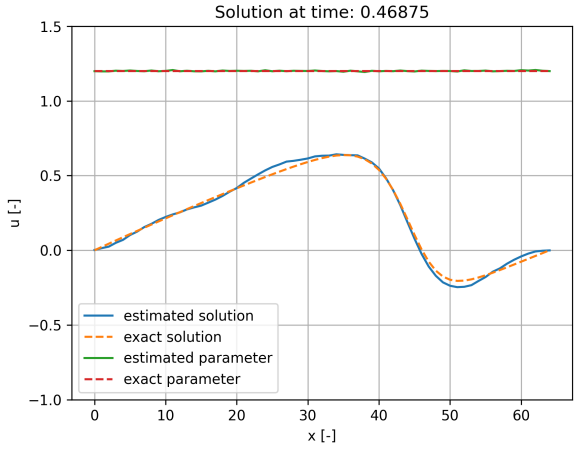

Solution at time: 0.234375
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter

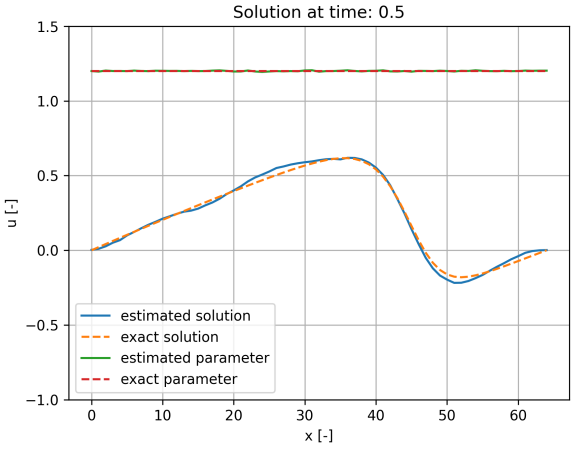

Solution at time: 0.28125
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter


Solution at time: 0.328125
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter


Solution at time: 0.375
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter


Solution at time: 0.421875
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter


Solution at time: 0.46875
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter


Solution at time: 0.5
u [-]
x [-]
estimated solution
exact solution
estimated parameter
exact parameter

Bibliography

[1] J. Adler and O. Oktem. Solving ill-posed inverse problems using iterative deep neural networks. *Inverse Problems*, 378, 2017.

[2] L. Bar and N. Sochen. Unsupervised deep learning algorithm for pde-based forward and inverse problems. *arXiv preprint arXiv:1904.05417*, 2019.

[3] H. Bateman. Some recent research on the motion of fluyds. *Monthly Weather Review*, 43(4):163–170, 1915.

[4] S. Bhatnagar, Y. Afshar, S. Pan, K. Duraisamy, and S. Kaushik. Prediction of aerodynamic flow fields using convolutional neural networks. *Computational Mechanics*, pages 1–21, 2019.

[5] K. Bhattacharya, B. Kovachki, and A. M. Stuart. Model reduction and neural networks for parametric pde(s). *preprint,*, 2020.

[6] F. Bouttier and P. Courtier. Data assimilation concepts and methos. *Meteorological Training Course Lecture Series*, 1999.

[7] J. M. Burgers. A mathematical model illustrating the theory of turbulences. *Advances in Applied Mechanics*, 1:171–199, 1948.

[8] W. E and B. Yu. The deep ritz method: A deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 3: 2194–6701, 2018.

[9] K. Gernaey, J. Huusom, and R. Gani. Developing surrogate models via computer based experiments. *12th International Symposium on Process Systems Engineering and 25th European Symposium on Computer Aided Process Engineering*, 2015.

[10] D. Greenfeld, G. Meirav, R. Barsi, I. Yavneh, and R. Kimmel. Learning to optimize multigrid pde solvers. *International Conference on Machine Learning*, pages 2415–2423, 2019.

[11] X. Guo, W. Li, and F. Iorio. Convolutional neural networks for steady flow ap-

proximation. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[12] C. M. Jiang, S. Esmaeilzadeh, K. Azizzadenesheli, K. Kashinath, M. Mustafa, H. A. Tchelepi, P. Marcus, and Anandkumar. Meshfreeflownet: A physics-constrained deep continuous space-time super-resolution framework. *aarXiv preprint arXiv:2005.01463*, 2020.

[13] M. Katzfuss, J. R. Stroud, and C. K. Wiklec. Understanding the ensemble kalman filter. *The american statistician*, 2016.

[14] Y. Khoo, J. Lu, and L. Ying. Solving parametric pde problems with artificial neural networks. *arXiv preprint arXiv:1707.03351*, 2017.

[15] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer. Machine learning accelerated computational fluid dynamics. *arXiv preprint arXiv:2102.01010*, 2021.

[16] S. Koziel, D. E. Ciaurri, and L. Leifsson. *Computational Optimizations, Methods and Algorithms*. Springler, 2011.

[17] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. M. Stuart, and A. Anandkumar. Neural operator: Graph kernel network for partial differential equations. *arXiv preprint arXiv:2003.03485*, 2020.

[18] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. Fourier neural operator for parametric partial differential equations. *ICLR*, 2021.

[19] 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.

[20] E. Mastinos. The kalman filter: a didactical overview. 2016.

[21] S. Misra, S. V. Ragurama, and M. K. Bobba. Relaxation system based sub-grid scale modelling for large eddy simulation of burgers' equation. *International Journal of Computational Fluid Dynamics*, 24(8):305–315, 2010.

[22] T. Musha and H. Higuci. Traffic current fluctuation and the burgers equation. *Japanese Journal of Applied Physics*, 17(5):305–315, 1978.

[23] N. H. Nelsen and A. M. Stuart. The random feature model for input-output maps between banach spaces. *arXiv preprint arXiv:2005.10224*, 2020.

[24] S. Pan, K. Duraisamy, and G. E. Karniadakis. Physics-informed probabilistic learning of linear embeddings of nonlinear dynamics with guaranteed stability. *SIAM Journal on Applied Dynamical Systems*, 19:480–509, 2020.

[25] G. R. Patel, N. A. Trask, M. A. Wood, and E. C. Cyr. A physics-informed operator regression framework for extracting data-driven continuum models. *Computer Methods in Applied Mechanics and Engineering*, 373:113–500, 2021.

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

[27] J. D. Smith, K. Azizzadenesheli, and Z. E. Ross. Eikonet: Solving the eikonal equation with deep neural networks. *arXiv preprint arXiv:2004.00361*, 2020.

[28] V. N. Vanpik. *Statistical Learning Theory.* Wiley-Interscience, 1998. Rui Wang, Karthik Kashinath, Mustafa Mustafa, Adrian Albert, and, 1998.

[29] Y. Zhu and N. Zabaras. Bayesian deep convolutional encoder–decoder networks for surrogate modeling and uncertainty quantification. *Journal of Computational Physics*, 2018.

A | Appendix A

A.1. Ensemble Kalman Filter Method

A.1.1. Code for Classic Prey-Predator

```
import numpy as np
import matplotlib.pyplot as plt
import prevision as prv
from filterpy.common import Q_discrete_white_noise

T = 200
Nt = 1000
dt = T/Nt
x0 = 1
y0 = 0.1
rho = 0.1
alpha = 0.25
beta = 0.2
gamma = 0.1
delta = 0.1

# Define the measurament function
def hx(x):
   return np.array([x[0], x[1]])

# Define the transition function
def fx(x, dt):
    F = np.array([[1 + alpha*dt - beta*dt*x[1] -rho*dt*x[0],
                   0],
                  [0,                                        1-gamma*
   dt + delta*dt*x[0]]])
    return np.dot(F, x)

# Define the covariance matrix
P = np.eye(2)
# Define the measurament noise
```

```
R = np.eye(2)
# Define the process noise
Q = Q_discrete_white_noise(dim=2, dt=dt, var=.01)

# Exact solution
x_ex = np.zeros((Nt+1,2))
tt=np.arange(0,T+dt,dt)
for i,t in enumerate(tt):
    if i==0:
        x_ex[i,:] = np.array([x0,y0])
    else:
        x_ex[i,0] = x_ex[i-1,0] + dt*x_ex[i-1,0]*(alpha-beta*x_ex[i-1,1]-rho*x_ex[i-1,0])
        x_ex[i,1] = x_ex[i-1,1] + dt*x_ex[i-1,1]*(-gamma+delta*x_ex[i-1,0])

# Define the data acquisition function
def get_sensor_reading(t):
    i = np.int32(t/dt)
    return x_ex[i,:]

# Create the model from library
f = prv.EnKF(dim_x=2, dim_z=2, f=fx, h=hx, get_data=get_sensor_reading, dt=dt, t0=0)
f.create_model(x0=np.array([x0, y0]), P=P, R=R, Q=Q, N=100)

# Predict/Update loop
x_hat = f.loop(T)

plt.figure()
plt.grid(True)
plt.plot(tt,x_hat[:,0], label='estimated prey')
plt.plot(tt,x_hat[:,1], label='estimated predators')
plt.plot(tt,x_ex[:,0], label='exact prey', linestyle='--')
plt.plot(tt,x_ex[:,1], label='exact predators', linestyle='--')
plt.legend()
plt.ylabel('Prey and Predators [-]')
plt.xlabel('Time [-]')

plt.figure()
plt.grid(True)
plt.plot(x_hat[:,0],x_hat[:,1], label='estimated trajectory')
plt.plot(x_ex[:,0],x_ex[:,1], label='exact trajectory', linestyle='--')
plt.legend()
```

```
plt.ylabel('Predators [-]')
plt.xlabel('Prey [-]')
```

A.1.2. Code for Prey-Predator with state-parameter

```
import numpy as np
import matplotlib.pyplot as plt
import prevision as prv
from filterpy.common import Q_discrete_white_noise

T = 200
Nt = 1000
dt = T/Nt
x0 = 1
y0 = 0.1
rho = 0.1
alpha = 0.25
beta = 0.2
gamma = 0.1
delta = 0.1

# Define the measurament function
def hx(x):
   return np.array([x[0], x[1]])

# Define the transition function
def fx(x, dt):
    F = np.array([[1 + x[2]*dt - beta*dt*x[1] -rho*dt*x[0],
                   0,                   0],
                  [0,                                                 1-gamma*
   dt + delta*dt*x[0],                   0],
                  [0,
                   0,                   1]])
    return np.dot(F, x)

# Assign the initial x array
x = np.array([x0, y0, 0.25])

# Define the covariance matrix
P = np.eye(3)
# Define the measurament noise
R = np.eye(2)
# Define the process noise
Q = Q_discrete_white_noise(dim=3, dt=dt, var=.00)
```

```
37
38 # Exact solution
39 x_ex = np.zeros((Nt+1,2))
40 tt=np.arange(0,T+dt,dt)
41 for i,t in enumerate(tt):
42     if i==0:
43         x_ex[i,:] = np.array([x0,y0])
44     else:
45         x_ex[i,0] = x_ex[i-1,0] + dt*x_ex[i-1,0]*(alpha-beta*x_ex[i
    -1,1]-rho*x_ex[i-1,0])
46         x_ex[i,1] = x_ex[i-1,1] + dt*x_ex[i-1,1]*(-gamma+delta*x_ex[i
    -1,0])
47
48 # Define the data acquisition function
49 def get_sensor_reading(t):
50     i = np.int32(t/dt)
51     return x_ex[i,:]
52
53 # Create the model from library
54 f = prv.EnKF(dim_x=3, dim_z=2, f=fx, h=hx, get_data=get_sensor_reading,
    dt=dt, t0=0)
55 f.create_model(x0=x, P=P, R=R, Q=Q, N=500)
56
57 # Predict/Update loop
58 x_hat = f.loop(T)
59
60 plt.figure()
61 plt.grid(True)
62 plt.plot(tt,x_hat[:,0], label='estimated prey')
63 plt.plot(tt,x_hat[:,1], label='estimated predators')
64 plt.plot(tt,x_ex[:,0], label='exact prey', linestyle='--')
65 plt.plot(tt,x_ex[:,1], label='exact predators', linestyle='--')
66 plt.legend()
67 plt.ylabel('Prey and Predators [-]')
68 plt.xlabel('Time [-]')
69
70 plt.figure()
71 plt.grid(True)
72 plt.plot(x_hat[:,0],x_hat[:,1], label='estimated trajectory')
73 plt.plot(x_ex[:,0],x_ex[:,1], label='exact trajectory', linestyle='--')
74 plt.legend()
75 plt.ylabel('Predators [-]')
76 plt.xlabel('Prey [-]')
77
```

```
plt.figure()
plt.grid(True)
plt.plot(tt,x_hat[:,2], label='estimated alpha')
plt.axhline(alpha, xmin=0, xmax=tt[-1], label='exact alpha', linestyle='
    --')
plt.legend()
plt.ylabel('Alpha coefficient [-]')
plt.xlabel('Time [-]')
```

A.1.3. Code for Classic Prey-Predator with surrogate model

```
import numpy as np
import matplotlib.pyplot as plt
import prevision as prv
from filterpy.common import Q_discrete_white_noise

T = 200
Nt = 200
dt = T/Nt
x0 = 1
y0 = 0.1
rho = 0.1
alpha = 0.25
beta = 0.2
gamma = 0.1
delta = 0.1

# Define the measurament function
def hx(x):
    return np.array([x[0], x[1]])

# Define the transition function
NN=keras.models.load_model('../data/Prey_Predator/model_NN.h5', compile=
    False)
def fxx(x, dt):
    x_ = np.empty((1,2,))
    x_[0,:] = x
    return NN(x_)

# Assign the initial x array
x = np.array([x0, y0])

# Define the covariance matrix
P = np.eye(2)
```

```
# Define the measurament noise
R = np.eye(2)
# Define the process noise
Q = Q_discrete_white_noise(dim=2, dt=dt, var=.01)

# Exact solution
x_ex = np.zeros((Nt+1,2))
tt=np.arange(0,T+dt,dt)
for i,t in enumerate(tt):
    if i==0:
        x_ex[i,:] = np.array([x0,y0])
    else:
        x_ex[i,0] = x_ex[i-1,0] + dt*x_ex[i-1,0]*(alpha-beta*x_ex[i-1,1]-rho*x_ex[i-1,0])
        x_ex[i,1] = x_ex[i-1,1] + dt*x_ex[i-1,1]*(-gamma+delta*x_ex[i-1,0])

# Define the data acquisition function
def get_sensor_reading(t):
    i = np.int32(t/dt)
    return x_ex[i,:]

# Create the model from library
f = prv.EnKF(dim_x=2, dim_z=2, f=fxx, h=hx, get_data=get_sensor_reading, dt=dt, t0=0)
f.create_model(x0=np.array([x0, y0]), P=P, R=R, Q=Q, N=100)

# Predict/Update loop
x_hat = f.loop(T, verbose=True)

plt.figure()
plt.grid(True)
plt.plot(tt,x_hat[:,0], label='estimated prey')
plt.plot(tt,x_hat[:,1], label='estimated predators')
plt.plot(tt,x_ex[:,0], label='exact prey', linestyle='--')
plt.plot(tt,x_ex[:,1], label='exact predators', linestyle='--')
plt.legend()
plt.ylabel('Prey and Predators [-]')
plt.xlabel('Time [-]')

plt.figure()
plt.grid(True)
plt.plot(x_hat[:,0],x_hat[:,1], label='estimated trajectory')
plt.plot(x_ex[:,0],x_ex[:,1], label='exact trajectory', linestyle='--')
```

```
74 plt.legend()
75 plt.ylabel('Predators [-]')
76 plt.xlabel('Prey [-]')
```

B | Appendix B

B.1. FNO model for Burgers equation

B.1.1. Code for generating training data

```
import numpy as np
from scipy import interpolate
from IPython.display import clear_output

# Training set
N_samples = 640
Nx = 2**6
Nt = 2**17
dx = 1/Nx
dz = 5*dx
dt = 1/Nt
xx = np.arange(0,1+dx,dx)
zz = np.arange(0,1+dz,dz)
tt = np.arange(0,1+dt,dt)
nu = 0.1
mean = 0
std_dev = 1
input_list = []
output_list =[]

for n in range(N_samples):
    clear_output(wait=True)
    print('Advancing: '+str(n/N_samples*100) +'%')
    u0_ = np.random.normal(mean, std_dev, zz.shape[0])
    u0_[0] = 0
    u0_[-1] = u0_[0]
    spl = interpolate.splrep(zz,u0_)
    u0 = interpolate.splev(xx,spl)
    uh = np.zeros((xx.shape[0],tt.shape[0]))
    uh[:,0] = u0
    for j in range(0, tt.shape[0]-1):
```

```
        for i in range(1, xx.shape[0]-1):
            uh[i,j+1] = uh[i,j] + nu*dt*(uh[i+1,j] - 2*uh[i,j] + uh[i-1,j])/(dx**2) - 0.5*dt*(uh[i,j]**2-uh[i-1,j]**2)/dx
    input_list.append(u0)
    output_list.append(uh[:,-1])

input_train = np.array(input_list)
output_train = np.array(output_list)

np.save('../data/Burgers/input_train.npy', input_train)
np.save('../data/Burgers/output_train.npy', output_train)
print('train dataset saved')

# Test set
N_samples = 160
Nx = 2**6
Nt = 2**17
dx = 1/Nx
dz = 5*dx
dt = 1/Nt
xx = np.arange(0,1+dx,dx)
zz = np.arange(0,1+dz,dz)
tt = np.arange(0,1+dt,dt)
nu = 0.1
mean = 0
std_dev = 1
input_list = []
output_list =[]

for n in range(N_samples):
    clear_output(wait=True)
    print('Advancing: '+str(n/N_samples*100) +'%')
    u0_ = np.random.normal(mean, std_dev, zz.shape[0])
    u0_[0] = 0
    u0_[-1] = u0_[0]
    spl = interpolate.splrep(zz,u0_)
    u0 = interpolate.splev(xx,spl)
    uh = np.zeros((xx.shape[0],tt.shape[0]))
    uh[:,0] = u0
    for j in range(0, tt.shape[0]-1):
        for i in range(1, xx.shape[0]-1):
            uh[i,j+1] = uh[i,j-1] + nu*dt*(uh[i+1,j-1] - 2*uh[i,j-1] + uh[i-1,j-1])/(dx**2) - dt*(uh[i,j]**2-uh[i-1,j]**2)/dx
    input_list.append(u0)
```

```
    output_list.append(uh[:,-1])

input_test = np.array(input_list)
output_test = np.array(output_list)

np.save('../data/Burgers/input_test.npy', input_test)
np.save('../data/Burgers/output_test.npy', output_test)
print('test dataset saved')
```

B.1.2. Code for implementing the FNO model

```
import numpy as np
import matplotlib.pyplot as plt
import prevision as prv
import tensorflow as tf

lr_schedule = tf.keras.optimizers.schedules.InverseTimeDecay(
  0.01,
  decay_steps=200,
  decay_rate=5,
  staircase=True)

def get_optimizer():
    return tf.keras.optimizers.Adam(lr_schedule)

def get_callbacks(name):
    return [
        tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience
    =2000, min_delta=0.0001),
        ]

def compile_and_fit(model, name, x_train, y_train, x_test, y_test,
    optimizer=None, max_epochs=1000):
    if optimizer is None:
        optimizer = get_optimizer()

    model.compile(optimizer=optimizer,
                  loss=tf.keras.losses.MeanSquaredError(),
                  metrics=[
                    tf.keras.losses.MeanSquaredError("auto", "
    mean_squared_error"),
                    'accuracy'])
    model.summary()

```

```
    history = model.fit(
            x_train, y_train,
            epochs=max_epochs,
            validation_data=(x_test, y_test),
            callbacks=get_callbacks(name),
            verbose=2)
    return history

# Loading training dataset
folder = 'Burgers'
input_train=np.load('../data/' + folder + '/input_train.npy')
input_test=np.load('../data/' + folder + '/input_test.npy')
output_train=np.load('../data/' + folder + '/output_train.npy')
output_test=np.load('../data/' + folder + '/output_test.npy')

INPUTDIM = (input_train.shape[1],)
OUTPUTDIM = (output_train.shape[1],)

# Creting model
model = prv.FNO(INPUTDIM, OUTPUTDIM, p_dim=256, n=11, k_max=7, verbose=
    True, model_name='Burgers_FNO', dropout=0.05, kernel_reg=0.005)
history = compile_and_fit(model, model.name, x_train=input_train,
    y_train=output_train, x_test=input_test, y_test=output_test,
    optimizer=None, max_epochs=10000)

bc     = history.history['mean_squared_error']
val_bc = history.history['val_mean_squared_error']

plt.loglog(range(np.shape(bc)[0]),bc, linewidth=2,)
plt.loglog(range(np.shape(val_bc)[0]),val_bc,'--', linewidth=2,)
plt.title('FNO learning')
plt.grid(True)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(('Loss training set','Loss test set'))
plt.show()

model.save('../data/'+folder+'/Burgers_FNO.h5')
```

B.1.3. Code for using the FNO model

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
```

```
from random import randint

input_test=np.load('../data/' + folder + '/input_test.npy')
output_test=np.load('../data/' + folder + '/output_test.npy')

folder = 'Burgers'
loaded_model = keras.models.load_model('../data/'+folder+'/Burgers_FNO.
    h5', compile=False)

rec_output = loaded_model.predict(input_test)
index = randint(0, input_test.shape[0]-1)
xx=np.linspace(0,1,rec_output.shape[1])

plt.figure()
plt.grid(True)
plt.plot(xx,rec_output[index,:], label='estimated solution')
plt.plot(xx,output_test[index,:], label='exact solution', linestyle='--'
    )
plt.legend()
plt.ylabel('u(x,t) [-]')
plt.xlabel('x [-]')
plt.title('Test set - index: '+ str(index))
```

B.2. FNO model for Darcy equation

B.2.1. Code for generating training data

```
import numpy as np
from neuralop.datasets import load_darcy_flow_small

n_train = 1000

train_loader, test_loaders, output_encoder = load_darcy_flow_small(
        n_train=n_train, batch_size=4,
        )

train_dataset = train_loader.dataset

input_list = []
output_list = []
for index in range(n_train):
    data = train_dataset[index]
    x = data['x']
```

```
    y = data['y']
    a = x[0].numpy()
    u = y[0].numpy()
    input_list.append(a)
    output_list.append(u)

input_train = np.array(input_list)[:800,:,:]
output_train = np.array(output_list)[:800,:,:]
input_test = np.array(input_list)[800:,:,:]
output_test = np.array(output_list)[800:,:,:]

np.save('../data/Darcy/input_train.npy', input_train)
np.save('../data/Darcy/output_train.npy', output_train)
print('train dataset saved')

np.save('../data/Darcy/input_test.npy', input_test)
np.save('../data/Darcy/output_test.npy', output_test)
print('test dataset saved')
```

B.2.2. Code for implementing the FNO model

```
import numpy as np
import matplotlib.pyplot as plt
import prevision as prv
import tensorflow as tf
from tensorflow import keras

lr_schedule = tf.keras.optimizers.schedules.InverseTimeDecay(
  0.01,
  decay_steps=50,
  decay_rate=6,
  staircase=True)

def get_optimizer():
    return tf.keras.optimizers.Adam(lr_schedule)

def get_callbacks(name):
    return [
        tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience
   =200, min_delta=0.0001),
        ]

def compile_and_fit(model, name, x_train, y_train, x_test, y_test,
   optimizer=None, max_epochs=1000):
```

```
22
23      if optimizer is None:
24          optimizer = get_optimizer()
25
26      model.compile(optimizer=optimizer,
27                    loss=tf.keras.losses.MeanSquaredError(),
28                    metrics=[
29                      tf.keras.losses.MeanSquaredError("auto", "
    mean_squared_error"),
30                      'accuracy'])
31
32      model.summary()
33
34      history = model.fit(
35              x_train, y_train,
36              epochs=max_epochs,
37              validation_data=(x_test, y_test),
38              callbacks=get_callbacks(name),
39              verbose=2)
40
41      return history
42
43  # Loading training dataset
44  folder = 'Darcy'
45  input_train=np.load('../data/' + folder + '/input_train.npy')
46  input_test=np.load('../data/' + folder + '/input_test.npy')
47  output_train=np.load('../data/' + folder + '/output_train.npy')
48  output_test=np.load('../data/' + folder + '/output_test.npy')
49
50  INPUTDIM = (input_train.shape[1],input_train.shape[2])
51  OUTPUTDIM = (output_train.shape[1],output_train.shape[2])
52
53  # Creting model
54  model = prv.FNO2D(INPUTDIM, OUTPUTDIM, p_dim=32, n=11, k_max=7, verbose=
    True, model_name='Darcy_FNO', dropout=0.05, kernel_reg=0.005)
55  history = compile_and_fit(model, model.name, x_train=input_train,
    y_train=output_train, x_test=input_test, y_test=output_test,
    optimizer=None, max_epochs=1000)
56
57  bc     = history.history['loss']
58  val_bc = history.history['val_loss']
59
60  plt.loglog(range(np.shape(bc)[0]),bc, linewidth=2,)
61  plt.loglog(range(np.shape(val_bc)[0]),val_bc,'--', linewidth=2,)
```

```
plt.title('FNO learning')
plt.grid(True)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(('Loss training set','Loss validation set'))
plt.show()

model.save('../data/'+folder+'/Darcy_FNO.h5')
```

B.2.3. Code for using the FNO model

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from random import randint

folder = 'Darcy'

input_test=np.load('../data/' + folder + '/input_test.npy')
output_test=np.load('../data/' + folder + '/output_test.npy')

loaded_model = keras.models.load_model('../data/'+folder+'/Darcy_FNO.h5'
    , compile=False)

from random import randint
rec_output = model.predict(input_test)
index = randint(0, input_test.shape[0]-1)

fig = plt.figure(figsize=(10, 10))

ax = fig.add_subplot(2, 2, 1)
ax.set_title('Exact solution')
m = ax.imshow(output_test[index])
fig.colorbar(m, ax=ax, location='right')

ax = fig.add_subplot(2, 2, 2)
ax.set_title('Reconstructed solution')
m = ax.imshow(rec_output[index])
fig.colorbar(m, ax=ax, location='right')

ax = fig.add_subplot(2, 2, 3)
ax.set_title('Error')
m = ax.imshow(output_test[index]-rec_output[index])
fig.colorbar(m, ax=ax, location='right')
```

```
33
34 ax = fig.add_subplot(2, 2, 4)
35 ax.set_title('Input')
36 m = ax.imshow(input_train[index])
37 fig.colorbar(m, ax=ax, location='right')
```

C | Appendix C

C.1. FNO model for Burgers equation in time

C.1.1. Code for generating training data

```
import numpy as np
from IPython.display import clear_output
from scipy import interpolate

Nx = 2**6
Nt = 2**18
dx = 1/Nx
dz = 4*dx
dt = 1/Nt
xx = np.arange(0,1+dx,dx)
zz = np.arange(0,1+dz,dz)
tt = np.arange(0,0.25+dt,dt)
nu = 0.025
resolution = 2048*2

def compute_u0(xx,zz, mean=0, std_dev=0):
    mean = 0
    std_dev = 1
    u0_ = np.random.normal(mean, std_dev, zz.shape[0])
    spl = interpolate.splrep(zz,u0_)
    u0 = interpolate.splev(xx,spl)
    u0[0] = 0
    u0[-1] = 0
    return u0

input_list = []
output_list = []
N_samples = 100
for n in range(N_samples):
    u0 = compute_u0(xx,zz)
    uh = np.zeros((xx.shape[0],tt.shape[0]))
```

```
    uh[:,0] = u0
    last_j=0
    for j in range(0, tt.shape[0]-1):
        for i in range(1, xx.shape[0]-1):
            uh[i,j+1] = uh[i,j] + nu*dt*(uh[i+1,j] - 2*uh[i,j] + uh[i-1,
   j])/(dx**2) - 0.5*dt*(uh[i,j]**2-uh[i-1,j]**2)/dx
        if np.mod(j,resolution)==0 and not(j==0):
            factor = np.amax(np.abs(uh[:,last_j]))
            input_list.append(1/factor*uh[:,last_j])
            output_list.append(1/factor*(uh[:,j]))
            last_j=j
    clear_output(wait=True)
    print('Advancing: '+str((n+1)/N_samples*100) +'%')

input = np.array(input_list)
output = np.array(output_list)

from sklearn.model_selection import train_test_split
input_train, input_test, output_train, output_test = train_test_split(
   input, output, test_size=0.2)

np.save('../data/Burgers_time/input_train.npy', input_train)
np.save('../data/Burgers_time/output_train.npy', output_train)
np.save('../data/Burgers_time/input_test.npy', input_test)
np.save('../data/Burgers_time/output_test.npy', output_test)
```

C.1.2. Code for implementing the FNO

```
import numpy as np
import matplotlib.pyplot as plt
import prevision as prv
import tensorflow as tf

lr_schedule = tf.keras.optimizers.schedules.InverseTimeDecay(
  0.01,
  decay_steps=100,
  decay_rate=2,
  staircase=True)

def custom_loss(y_true, y_pred):
    return tf.keras.losses.cosine_similarity(y_true,y_pred) + 30*tf.
   keras.losses.huber(y_true,y_pred)

def get_optimizer():
```

```
16      return tf.keras.optimizers.Adam(lr_schedule)
17
18  def get_callbacks(name):
19      return [
20          tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience
    =500, min_delta=0.0001),
21          ]
22
23  def compile_and_fit(model, name, x_train, y_train, x_test, y_test,
    optimizer=None, max_epochs=1000, batch_size=32):
24
25      if optimizer is None:
26          optimizer = get_optimizer()
27
28      model.compile(optimizer=optimizer,
29                    loss=custom_loss,
30                    metrics=[
31                      tf.keras.losses.MeanSquaredError("auto", "
    mean_squared_error")])
32
33      model.summary()
34
35      history = model.fit(
36              x_train, y_train,
37              epochs=max_epochs,
38              validation_data=(x_test,y_test),
39              callbacks=get_callbacks(name),
40              batch_size = batch_size,
41              verbose=2)
42
43      return history
44
45  # Loading training dataset
46  folder = 'Burgers_time'
47  input_train=np.load('../data/' + folder + '/input_train.npy')
48  input_test=np.load('../data/' + folder + '/input_test.npy')
49  output_train=np.load('../data/' + folder + '/output_train.npy')
50  output_test=np.load('../data/' + folder + '/output_test.npy')
51
52  INPUTDIM = (input_train.shape[1],)
53  OUTPUTDIM = (output_train.shape[1],)
54
55  # Creting model
56  model = prv.FNO(INPUTDIM, OUTPUTDIM, p_dim=512, n=3, k_max=17, verbose=
```

```
    True, model_name='Burgers_time_FNO', dropout=0.01, kernel_reg=0.001)
history = compile_and_fit(model, model.name, x_train=input_train,
    y_train=output_train, x_test=input_test, y_test=output_test,
    batch_size=60 , max_epochs=5000)

bc     = history.history['loss']
val_bc = history.history['val_loss']

plt.semilogx(range(np.shape(bc)[0]),bc, linewidth=2,)
plt.semilogx(range(np.shape(val_bc)[0]),val_bc,'--', linewidth=2,)
plt.title('FNO learning')
plt.grid(True)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.ylim([-1,0])
plt.legend(('Loss training set','Loss validation set'))
plt.show()

model.save('../data/'+folder+'/Burgers_time_FNO.h5')
```

C.1.3. Code for using the FNO model

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from random import randint

folder = 'Burgers_time'

input_test=np.load('../data/' + folder + '/input_test.npy')
output_test=np.load('../data/' + folder + '/output_test.npy')

loaded_model = keras.models.load_model('../data/'+folder+'/
    Burgers_time_FNO.h5', compile=False)

rec_output = loaded_model.predict(input_test)
index = randint(0, input_test.shape[0]-1)
xx=np.linspace(0,1,rec_output.shape[1])

plt.figure()
plt.grid(True)
plt.plot(xx,rec_output[index,:], label='estimated solution')
plt.plot(xx,output_test[index,:], label='exact solution', linestyle='--'
    )
```

```
plt.legend()
plt.ylabel('u(x,t) [-]')
plt.xlabel('x [-]')
plt.title('Test set - index: '+ str(index))
```

C.1.4. Code for implementing the EnKF model

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
import prevision as prv

folder = 'Burgers_time'

Nx = 2**6
Nt = 2**18
dx = 1/Nx
dz = 4*dx
dt = 1/Nt
xx = np.arange(0,1+dx,dx)
zz = np.arange(0,1+dz,dz)
tt = np.arange(0,0.5+dt,dt)
nu = 0.025
u0 = np.load('../Data/'+ folder +'/u0.npy')
resolution = Nt/64

def compute_ex_sol(xx, tt, u0, nu, resolution):
    u=[]
    uh = np.zeros((xx.shape[0],tt.shape[0]+1))
    uh[:,0] = u0
    for j in range(0, tt.shape[0]):
        for i in range(1, xx.shape[0]-1):
            uh[i,j+1] = uh[i,j] + nu*dt*(uh[i+1,j] - 2*uh[i,j] + uh[i-1,
    j])/(dx**2) - 0.5*dt*(uh[i,j]**2-uh[i-1,j]**2)/dx
        if j==0:
            u.append(u0)
        elif np.mod(j,resolution)==0:
            u.append(uh[:,j])
    return np.array(u)

# Exact solution
u_ex = compute_ex_sol(xx,tt,u0,nu,resolution)

T = 0.5
```

```
Nt = 32
dt = T/Nt
dim = Nx+1

# Define the measurament function
def hx(x):
   return x

# Define the transition function
FNO=keras.models.load_model('../data/' + folder +'/Burgers_time_FNO.h5',
    compile=False)
def fxx(u, dt):
    factor = np.amax(np.abs(u))
    return factor*(FNO(np.array([u,]))[0])

# Define the covariance matrix
P = np.cov(u_ex, rowvar=False)
# Define the measurament noise
R = 0.1*np.eye(dim)
# Define the process noise
Q = 0.1*np.eye(dim)

# Define the data acquisition function
def get_sensor_reading(t):
    i = np.int32(t/dt)
    return u_ex[i,:]

# Create the model from library
f = prv.EnKF(dim_x=dim, dim_z=dim, f=fxx, h=hx, get_data=
   get_sensor_reading, dt=dt, t0=0)
f.create_model(x0=u0, P=P, R=R, Q=Q, N=10000)

# Predict/Update loop
u_hat = f.loop(T, verbose=True)

plt.figure()
for t_index in range(0,tt.shape[0]):
    plt.title('Solution at time: '+str(t_index*dt))
    plt.grid(True)
    plt.plot(u_hat[t_index,:], label='estimated solution')
    plt.plot(u_ex[t_index,:], label='exact solution', linestyle='--')
    plt.xlabel('x [-]')
    plt.ylabel('u [-]')
    plt.ylim([-1,1])
```

```
    plt.legend(loc='upper right')
    plt.savefig('../Burgers_time_EnKF_' + str(t_index) + '.png', dpi
    =300)
    plt.clf()
```

C.2. FNO model for Burgers equation in time with a state-parameter

C.2.1. Code for generating training data

```
import numpy as np
from IPython.display import clear_output
from scipy import interpolate

Nx = 2**6
Nt = 2**18
dx = 1/Nx
dz = 8*dx
dt = 1/Nt
xx = np.arange(0,1+dx,dx)
zz = np.arange(0,1+dz,dz)
tt = np.arange(0,0.25+dt,dt)
nu = 0.025
resolution = np.int32(Nt/64)

def compute_u0(xx,zz, mean=0, std_dev=0.5):
    mean = 0
    std_dev = 1
    u0_ = np.random.normal(mean, std_dev, zz.shape[0])
    u0_[0] = 0
    u0_[-1] = 0
    spl = interpolate.splrep(zz,u0_)
    u0 = interpolate.splev(xx,spl)
    u0[0] = 0
    u0[-1] = 0
    return u0

input_list = []
output_list = []
N_samples = 50
for n in range(N_samples):
    u0 = compute_u0(xx,zz)
```

```
    uh = np.zeros((xx.shape[0],tt.shape[0]))
    uh[:,0] = u0
    last_j=0
    for j in range(0, tt.shape[0]-1):
        cx = np.random.uniform(0.5,1.5,1)*np.ones_like(u0)
        for i in range(1, xx.shape[0]-1):
            uh[i,j+1] = uh[i,j] + nu*dt*(uh[i+1,j] - 2*uh[i,j] + uh[i-1,
   j])/(dx**2) - 0.5*cx[i-1]*dt*(uh[i,j]**2-uh[i-1,j]**2)/dx
        if np.mod(j,resolution)==0 and not(j==0):
            factor = np.amax(np.abs(uh[:,last_j]))
            input_list.append([1/factor*uh[:,last_j], 1/factor*cx])
            output_list.append(1/factor*(uh[:,j]))
            last_j=j
    clear_output(wait=True)
    print('Advancing: '+str((n+1)/N_samples*100) +'%')

input = np.array(input_list)
output = np.array(output_list)

from sklearn.model_selection import train_test_split
input_train, input_test, output_train, output_test = train_test_split(
   input, output, test_size=0.2)

np.save('../data/Burgers_time_cx/input_train.npy', input_train)
np.save('../data/Burgers_time_cx/output_train.npy', output_train)
np.save('../data/Burgers_time_cx/input_test.npy', input_test)
np.save('../data/Burgers_time_cx/output_test.npy', output_test)
```

C.2.2. Code for implementing the FNO

```
import numpy as np
import matplotlib.pyplot as plt
import prevision as prv
import tensorflow as tf

lr_schedule = tf.keras.optimizers.schedules.InverseTimeDecay(
  0.01,
  decay_steps=100,
  decay_rate=8,
  staircase=True)

def custom_loss(y_true, y_pred):
    return tf.keras.losses.cosine_similarity(y_true,y_pred) + 10*tf.
   keras.losses.huber(y_true,y_pred)
```

```
14
15 def get_optimizer():
16     return tf.keras.optimizers.Adam(lr_schedule)
17
18 def get_callbacks(name):
19     return [
20         tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience
    =1000, min_delta=0.001),
21         ]
22
23 def compile_and_fit(model, name, x_train, y_train, x_test, y_test,
    optimizer=None, max_epochs=1000, batch_size=32):
24
25     if optimizer is None:
26         optimizer = get_optimizer()
27
28     model.compile(optimizer=optimizer,
29                 loss=custom_loss,
30                 metrics=[
31                   tf.keras.losses.MeanSquaredError("auto", "
    mean_squared_error")])
32
33     model.summary()
34
35     history = model.fit(
36             x_train, y_train,
37             epochs=max_epochs,
38             validation_data=(x_test,y_test),
39             callbacks=get_callbacks(name),
40             batch_size = batch_size,
41             verbose=2)
42
43     return history
44
45 # Loading training dataset
46 folder = 'Burgers_time_cx'
47 input_train=np.load('../data/' + folder + '/input_train.npy')
48 input_test=np.load('../data/' + folder + '/input_test.npy')
49 output_train=np.load('../data/' + folder + '/output_train.npy')
50 output_test=np.load('../data/' + folder + '/output_test.npy')
51
52 INPUTDIM = (input_train.shape[1],input_train.shape[2])
53 OUTPUTDIM = (output_train.shape[1],)
54
```

```
verbose = True
p_dim = 512+128
n=3
k_max = 14

kernel_reg = 0.0001
dropout = 0.005

# Creting model
input_layer = layers.Input(shape = INPUTDIM, name= 'input_layer')
input_layer_flat = layers.Reshape((INPUTDIM[0]*INPUTDIM[1],)) (
    input_layer)
P_layer = layers.Dense(p_dim, activation='relu', kernel_regularizer =
    regularizers.l2(kernel_reg), name='P_layer') (input_layer_flat)
P_layer = layers.Dropout(dropout) (P_layer)
# Repeat the custom module 'n' times
for i in range(n):
    if verbose:
        print('Creating Fourier Layer ' +str(i))
    if i ==0:
        fourier_module_output = prv.Fourier_Layer(name='fourier_layer_'+
    str(i), k_max=k_max)(P_layer)
    else:
        fourier_module_output = prv.Fourier_Layer(name='fourier_layer_'+
    str(i), k_max=k_max)(fourier_module_output)

output_layer = layers.Dense(OUTPUTDIM[0], activation='linear',
    kernel_regularizer = regularizers.l2(kernel_reg), name='output_layer'
    ) (fourier_module_output)
output_layer= layers.Dropout(dropout) (output_layer)

if verbose:
    print('----------------------------------------------------------')
model = tf.keras.Model(inputs=input_layer, outputs = output_layer, name
    = 'Burgers_time_FNO')
if verbose:
    model.summary()

history = compile_and_fit(model, model.name, x_train=input_train,
    y_train=output_train, x_test=input_test, y_test=output_test,
    batch_size=256 , max_epochs=5000)

bc     = history.history['loss']
val_bc = history.history['val_loss']
```

```
90
91 plt.semilogx(range(np.shape(bc)[0]),bc, linewidth=2,)
92 plt.semilogx(range(np.shape(val_bc)[0]),val_bc,'--', linewidth=2,)
93 plt.title('FNO learning')
94 plt.ylim([-1,1])
95 plt.grid(True)
96 plt.xlabel('Epochs')
97 plt.ylabel('Loss')
98 plt.legend(('Loss training set','Loss validation set'))
99 plt.show()
```

C.2.3. Code for implementing the EnKF model

```
1  folder = 'Burgers_time_cx'
2
3  Nx = 2**6
4  Nt = 2**18
5  dx = 1/Nx
6  dz = 4*dx
7  dt = 1/Nt
8  xx = np.arange(0,1+dx,dx)
9  zz = np.arange(0,1+dz,dz)
10 tt = np.arange(0,0.5+dt,dt)
11 nu = 0.025
12 u0 = np.load('../Data/Burgers_time/u0.npy')
13 resolution = Nt/64
14
15 def compute_ex_sol(xx, tt, u0, nu, resolution):
16     u=[]
17     uh = np.zeros((xx.shape[0],tt.shape[0]+1))
18     uh[:,0] = u0
19     for j in range(0, tt.shape[0]):
20         cx = 1.2*np.ones_like(u0)
21         for i in range(1, xx.shape[0]-1):
22             uh[i,j+1] = uh[i,j] + nu*dt*(uh[i+1,j] - 2*uh[i,j] + uh[i-1,
    j])/(dx**2) - 0.5*cx[i-1]*dt*(uh[i,j]**2-uh[i-1,j]**2)/dx
23         if j==0:
24             u.append(np.concatenate((u0,cx), axis=None))
25         elif np.mod(j,resolution)==0:
26             u.append(np.concatenate((uh[:,j],cx), axis=None))
27     return np.array(u)
28
29 # Exact solution
30 u_ex = compute_ex_sol(xx,tt,u0,nu,resolution)
```

```
x0 = np.concatenate((u0, 0.95*1.2*np.ones_like(u0)), axis=None)

T = 0.5
Nt = 32
dt = T/Nt
dim = 2*(Nx+1)

# Define the measurament function
def hx(x):
   return x

# Define the transition function
FNO=keras.models.load_model('../data/' + folder +'/Burgers_time_cx_FNO.
   h5', compile=False)
def fxx(u, dt):
    d = int(dim/2)
    factor = np.amax(np.abs(u[:d]))
    return np.concatenate((factor*(FNO(np.array([[u[:d], u[d:]],]))[0]),
   u[d:]), axis=None)

    # Define the covariance matrix
P = np.cov(u_ex, rowvar=False)
# Define the measurament noise
R = 0.1*np.eye(dim)
# Define the process noise
Q = 0.1*np.eye(dim)

# Define the data acquisition function
def get_sensor_reading(t):
    i = np.int32(t/dt)
    return u_ex[i,:]

# Create the model from library
f = prv.EnKF(dim_x=dim, dim_z=dim, f=fxx, h=hx, get_data=
   get_sensor_reading, dt=dt, t0=0)
f.create_model(x0=x0, P=P, R=R, Q=Q, N=10000)

# Predict/Update loop
u_hat = f.loop(T, verbose=True)

d = int(dim/2)
plt.figure()
for t_index in range(0,u_ex.shape[0]):
    plt.title('Solution at time: '+str(t_index*dt))
```

```
72      plt.grid(True)
73      plt.plot(u_hat[t_index,:d], label='estimated solution')
74      plt.plot(u_ex[t_index,:d], label='exact solution', linestyle='--')
75      plt.plot(u_hat[t_index,d:], label='estimated parameter')
76      plt.plot(u_ex[t_index,d:], label='exact parameter', linestyle='--')
77      plt.xlabel('x [-]')
78      plt.ylabel('u [-]')
79      plt.ylim([-1,1.5])
80      plt.legend(loc='lower left')
81      plt.savefig('../Burgers_time_cx_EnKF_' + str(t_index) + '.png', dpi
        =300)
82      plt.clf()
```

List of Figures

Acknowledgements

Thanks to Prof. Stefano Pagani, who guided us in this project and motivated us to explore these topics.

Adriano, Giacomo.