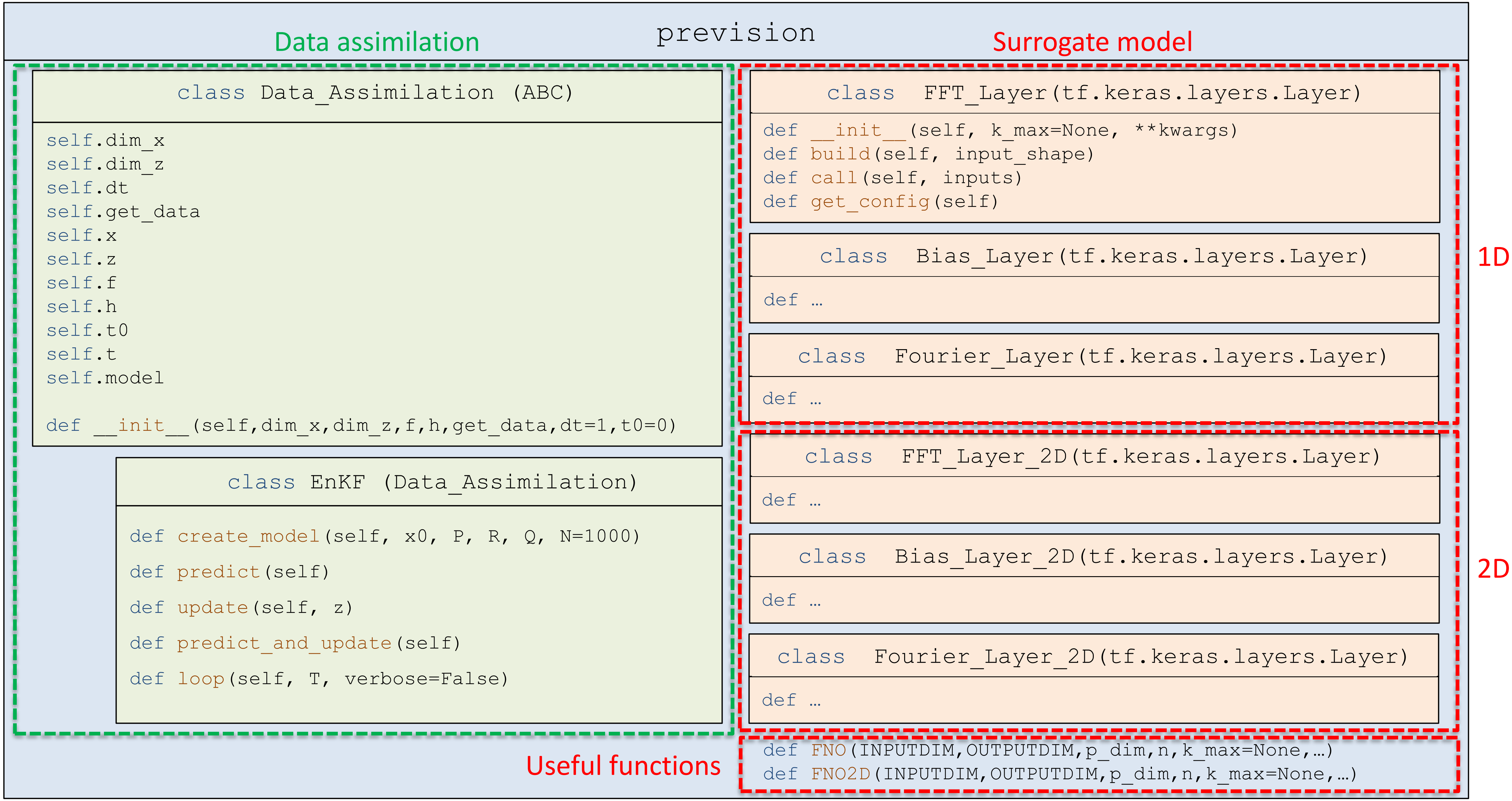
Coupling Ensemble Kalman Filter and Fourier Neural Operators for a comprehensive data assimilation based prediction model

Project for the course Advanced Programming for Scientific Computing

Adriano Minora
Giacomo Mondello Malvestiti

Contents

Library Overview



Data Assimilation via EnKF

Data assimilation via EnKF

Data assimilation is a mathematical time-stepping procedure used for determining the optimal state estimate of a system and that well interpolate sparse information data.

It is the result of a forecast based on previous step conditions and then corrected with observed data and estimated errors.

$$x_a = x_b + \delta x$$

δx represents the correction applied to vector x_b ,which is the background estimate of the true state, to find vector x_a that has to be the closest possible to the true state at the time of analysis.

Kalman filters methods

Kalman filters models are methods that aim to **predict** the true state of the phenomenon and to **correct** the prediction made.

The first phase is represented by the **apriori estimate**, which is the prediction based on the estimate of system last state.

The second phase represents the **aposteriori estimate** and it involves the correction of the first prediction made considering current time measurements.

The last estimate is the more representative of the system and the reliability of the method is directly dependent on the correction made.

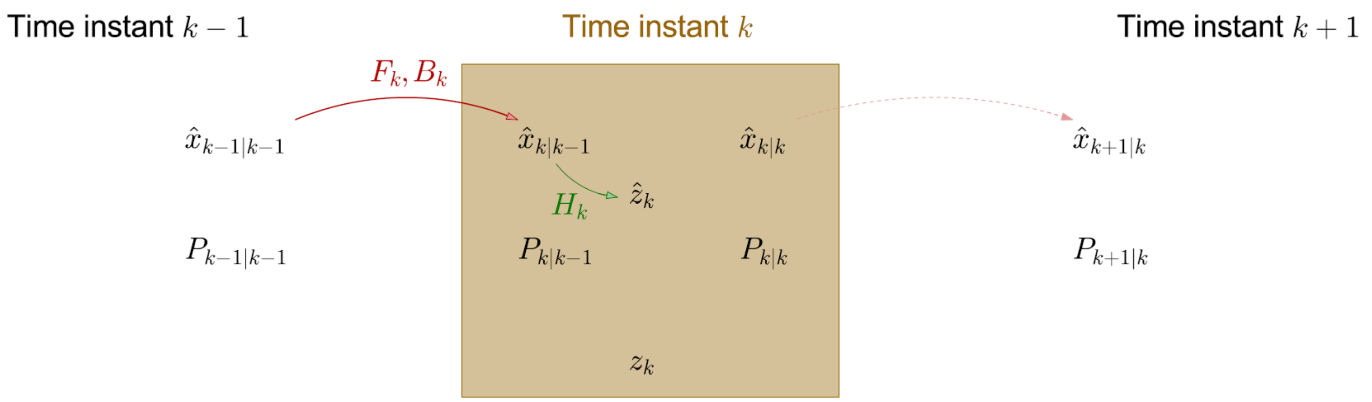
Basic Kalman Filter – Prediction phase

The first phase is governed by the theoretical evolution of the state vector x from time $k-1$ to time k.

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k$$

Where F_k is the state transition matrix and B_k is the control input matrix and the apriori Covariance matrix reads:

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k$$



Basic Kalman Filter – Correction phase

The aposteriori state estimate and the aposteriori covariance matrix are computed. The relation between the measurement and the apriori state vector is:

$$\hat{z}_k = H_k \hat{x}_{k|k-1} \qquad \text{with } H_k \text{ state to measurement transformation matrix.}$$

It is now possible to compute the optimal gain K

$$K_k = P_{k|k-1} H_k^T S_k^{-1} \qquad \text{with } S_k \text{ covariance of residuals } \tilde{z}_k = z_k - \hat{z}_k$$

Aposteriori estimate and covariance are:

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{z}_k$$

$$P_{k|k} = (I - K_k H_k) P_{k|k-1}$$

Basic Kalman Filters give good predictions when:

- the dynamics of the system is known and it is linear
- the noise is Gaussian
- the covariance matrices are known.

Ensemble Kalman Filter is capable of dealing with highly non-linear system with various types of noise.

Ensemble Kalman Filter

The Ensemble Kalman Filter is an extension of the Kalman Filter.

Basic Kalman Filter deals with the exact distribution of the state

Ensemble Kalman Filter works with an ensemble of vectors approximating the state distribution.

Predictions and corrections are made on N vectors which compose the ensemble.

$$x_k^{(i)} = F_k x_{k-1}^{(i)} + B_k u_k^{(i)} + q_k^{(i)}$$

$$\hat{z}_k^{(i)} = H_k \hat{x}_k^{(i)} + r_k^{(i)}$$

Implementation of EnKF model

Data assimiltion in phyton

- **Data Assimilation**: it is the abstract class that creates the object Data_assimilation eith all the parameters needed to develop the chosen method.

- **Abstract class** (ABC module): many assimilations models could be implemented. In this work the model chosen is **EnKF**.

- **EnKF**: child class that extends the general one by implementing the Ensamble Kalman filter method.

Data assimilation abstract class

```
class Data_Assimilation(ABC):
    def __init__(self, dim_x, dim_z, f, h, get_data, dt=1, t0=0):
        self.dim_x = dim_x
        self.dim_z = dim_z
        self.dt = dt
        self.get_data = get_data
        self.x = np.zeros((dim_x,))
        self.z = np.zeros((dim_z,))
        self.f = f
        self.h = h
        self.t0 = t0
        self.t = t0
        self.model = None
```

This is the abstract class made of a constructor that provides a generic structure for data assimilation algorithms. The essential parameters are initialized for the system.

EnKF child class – create model

```
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
```

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 this model more input data are necessary.

EnKF child class – methods

```python
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)
```

`Predict` and `update` method are two already existing methods present in the library `filterpy.kalman.EnsembleKalmanFilter`. The first one advances the state estimation and the second one updates the state estimation thanks to new observational data.

`Predict_and_update` combine the two together.

EnKF child class – methods

```python
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 function executes the function `predict_and_update` in a specific time range which depends on the final time T.

This function returns the estimated state over time `x_hat`.

EnKF Numerical Examples

Prey - Predator

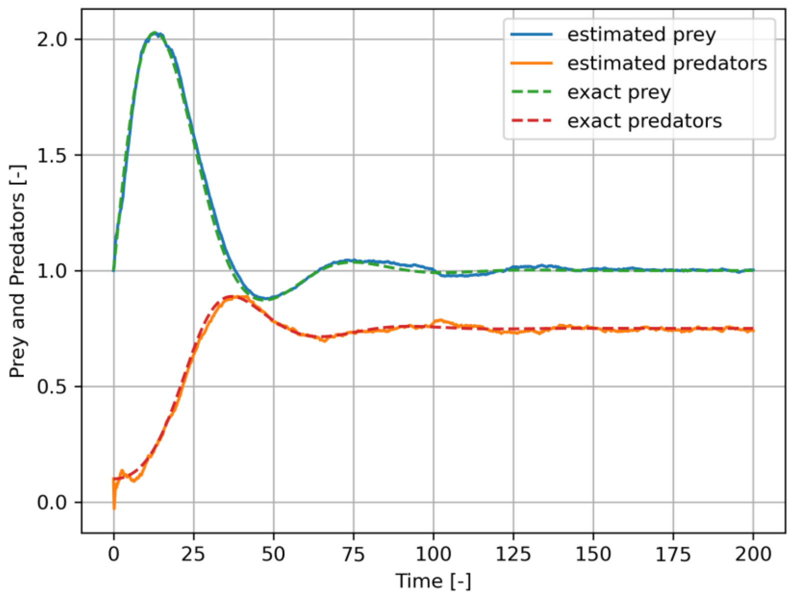
The model is represented by two differential equations that describe the rates of change of prey and predator population sizes over time:

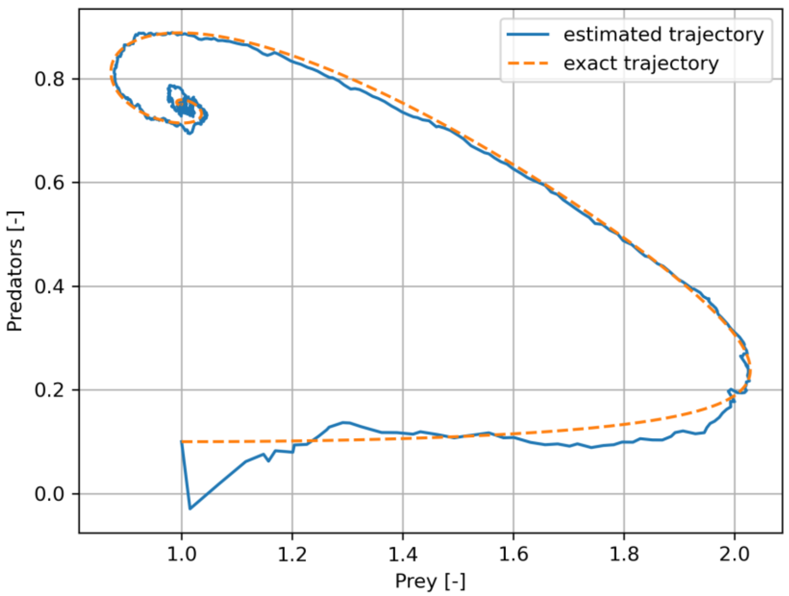
$$\frac{dx}{dt} = \alpha x - \beta xy - \rho x^2$$

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

Parameters α and ρ depends on the prey population, γ on the predator one, while β and δ are interaction terms between the two population

Classic Prey - Predator





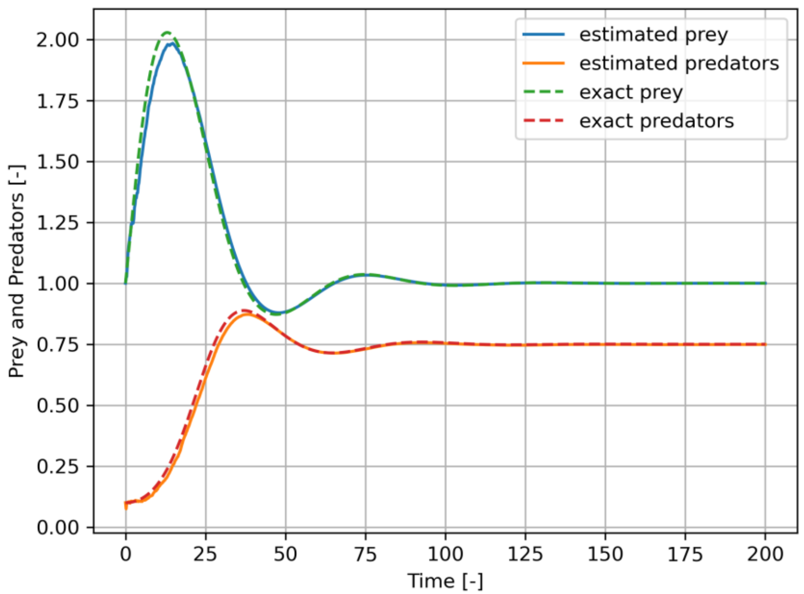
Transition function: governing equations of the phenomenon

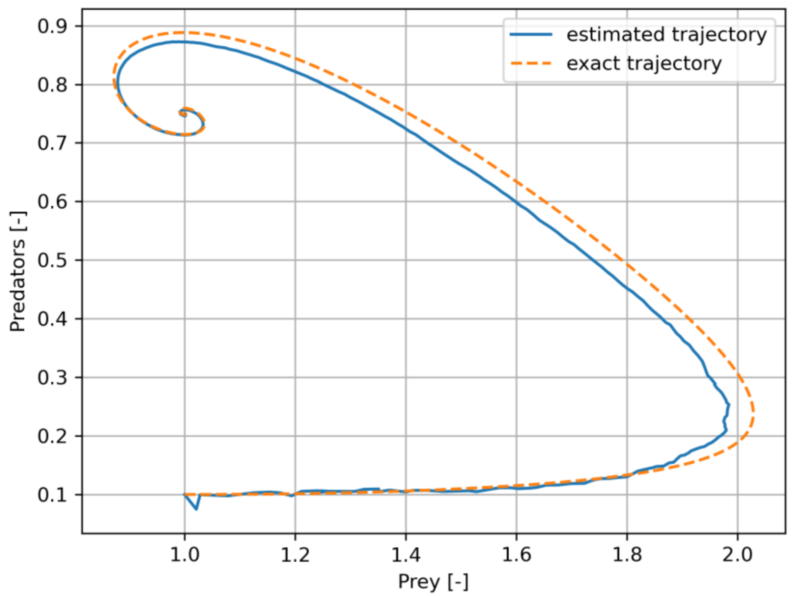
Measurement function: gives the state variables.

Covariance matrix and **measurement noise matrix** : identity matrix.

The observed data are taken by function `get_sensor_reading`.

Prey – Predator with state parameter





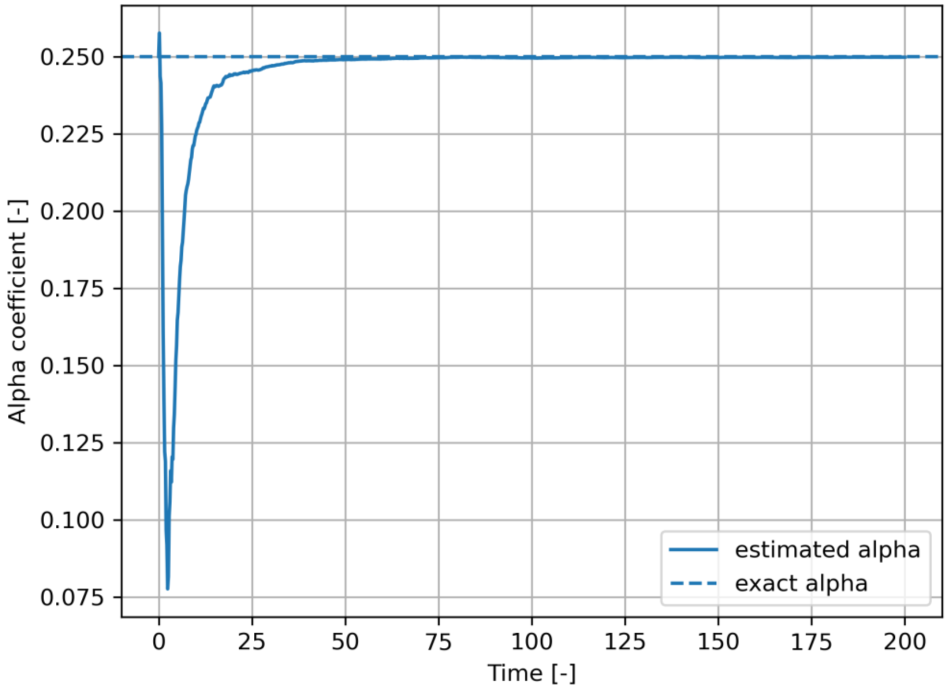
Transition function: governing equations of the phenomenon and constant one for the state parameter α.

Measurement function: gives the state variables.

Covariance matrix and **measurement noise matrix** : identity matrix.

The observed data are taken by function `get_sensor_reading`.

Prey – predator with state parameter

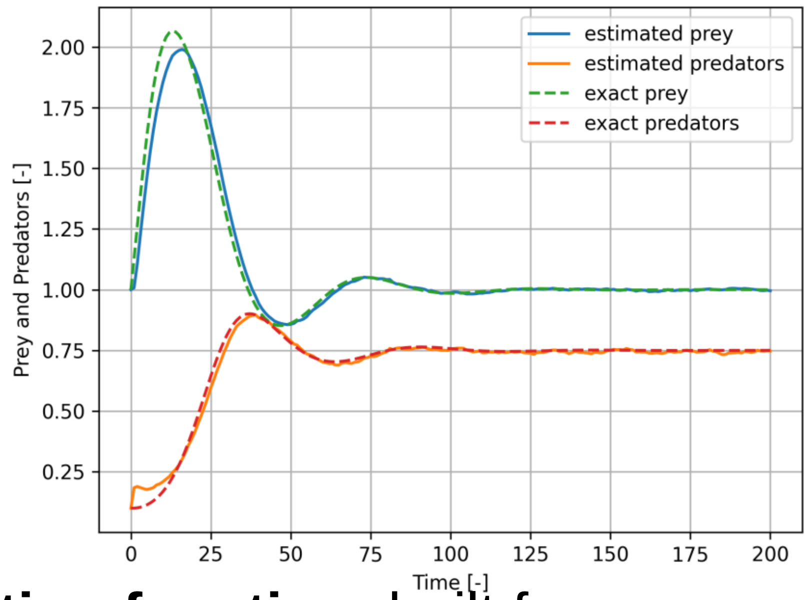


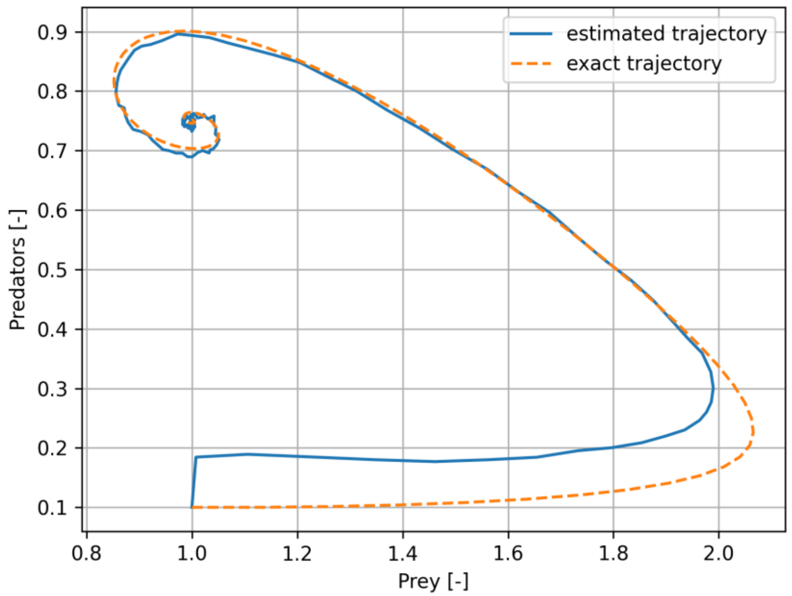
The parameter α becomes a state variable not governed by a differential equation and taken constant in time.

This problem simulates the case when one of the parameters of the phenomenon is not known and becomes then a variable.

The state parameter α is not taken into consideration by the measurement function because there is no way of measuring this value.

Prey – predator with surrogate model





Transition function: built from a neural network.

Measurement function: gives the state variables.

Covariance matrix and **measurement noise matrix** : identity matrix.

The observed data are taken by function `get_sensor_reading`.

Surrogate Model via FNO

Surrogate Model via FNO

- **Surrogate models**: simplified analytical representations designed to emulate the input/output behaviour of intricate systems.

- **Neural operators**: mitigate the mesh-dependent nature of finite-dimensional operator methods by producing a single set of network parameters usable with various discretizations.

- **Fourier Neural Operators**: Neural operators that rely on the Fourier transform as kernel integral operator

Learning operators

Consider two Banach spaces $\mathcal{A}$ and $\mathcal{U}$.

$\mathcal{A}$: space of inputs of a PDE

$\mathcal{U}$: space of outputs of a PDE

We want to learn a generic non-linear mapping $G^{\dagger}$ representing the PDE:

$$G^{\dagger}: \mathcal{A} \to \mathcal{U}$$

We can approximate the operator by a parametrization $G_{\theta}: \mathcal{A} \to \mathcal{U}$ with $\theta \in \Theta$.

The goal is to find $\theta^{\dagger} \in \Theta$ such that $G_{\theta^{\dagger}} \approx G^{\dagger}$

Learning Operators

Neural Operators can be formulated as an iterative architecture:

$$\mathcal{A} \ni a \rightarrow \underbrace{v_0 \rightarrow v_1 \rightarrow \ \ldots \rightarrow v_T}_{G_\theta} \rightarrow u \in \mathcal{U}$$

Input layer: $v_0(x) = P\big(a(x)\big)$ with P being a dense layer

Output layer: $u(x) = Q\big(v_T(x)\big)$ with Q being a dense layer

Iterative update: $v_{t+1}(x) = \sigma\left(W v_t(x) + \mathcal{K}_\phi v_t(x)\right)$ with $\phi \in \Theta_{\mathcal{K}}$

σ: non-linear activation funciton $\mathcal{K}_\phi$: kernel integral operator W: dense layer

Fourier Neural Operator

General kernel integral operator:

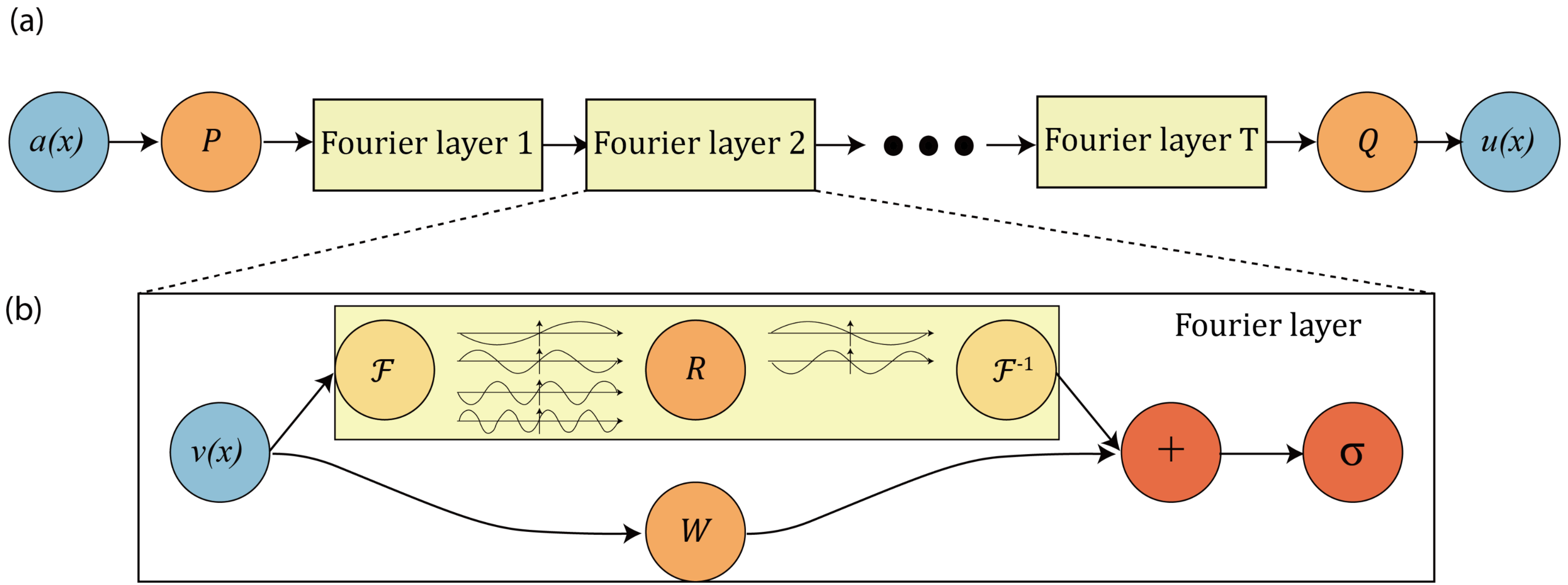
$$(\mathcal{K}_\phi v)(x) = \int_D k_\phi\big(x, y, a(x), a(y)\big) v(x)\, dy \quad \forall x \in D$$

Kernel integral operator for FNO:

$$\mathcal{K}_\phi v = \mathcal{F}^{-1}\Big(\mathcal{F}\big(k_\phi\big) \cdot \mathcal{F}(v)\Big) = \mathcal{F}^{-1}\Big(R_\phi \cdot \mathcal{F}(v)\Big)$$

R_ϕ: linear transformation in the Fourier space → dense layer in the Fourier space

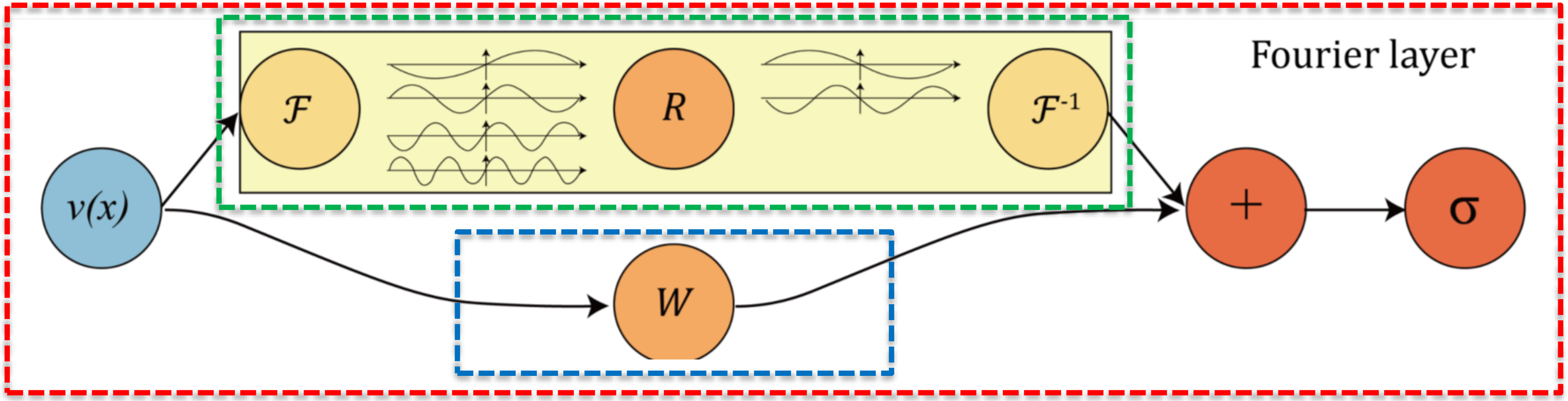
Fourier Neural Operator



Implementation of FNO model

1D FNO model in phyton

- **FFT Layer:** performs the Fast Fourier Transform on the inputs, applies the linear transformation in the Fourier space, transforms back into the inputs space.
- **Bias Layer:** applies a linear transformation in the inputs space.
- **Fourier Layer:** combines the two previous layers with a sum and then applies the non-linear activation function.



FFT Layer class – constructor

- Child class of `tf.keras.layers.Layer`
- The defined constructor `__init__` overrides the parent constructor.
- It takes as input the number of modes to be considered in the FFT and initializes the two variables representing the shape after the FFT and the inverse FFT.

```
@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
```

FFT Layer class – build method

```python
    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
        )
```

- Given a shape of the input array, 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`.

FFT Layer class – call method

- 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
```

FFT Layer class – configurations and properties

```python
    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
```

- The `get_config` method is needed to correctly save to file a `keras` layer with custom inputs.
- The shapes of the tensors after the convolution is needed outside the class to create the Bias Layer, so two properies have been defined.

Bias Layer class

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

```python
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()))
```

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

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

- The constructor takes as input an object of type FFT Layer, define by the previous class.
- The `build` method creates the bias tensor of the correct shape.
- The `call` method applies the linear transformation.
- The `get_config` method specifies the custom inputs of the layer

Fourier Layer class

```python
@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
```

- The constructor takes as input the number of modes in the FFT
- The `call` method call the previous classes and sum the tensors and apply the non-linear activation function.
- The `get_config` method specifies the custom inputs of the layer

The FNO function

```python
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 uses the previous classes to completely build the FNO.
- The user has to specify the dimensions of inputs and outputs, the dimension of the Fourier space (`p_dim`), the number of Fourier Layers (`n`) and optionally the number of modes in the FFT (`k_max`)

FNO Numerical Examples

1D – Burgers equation

Consider the Burgers equation in $\Omega = (0,1) \times (0,1]$ with $u_0 \in L^2(0,1;\mathbb{R})$ and $\nu = 0.1$

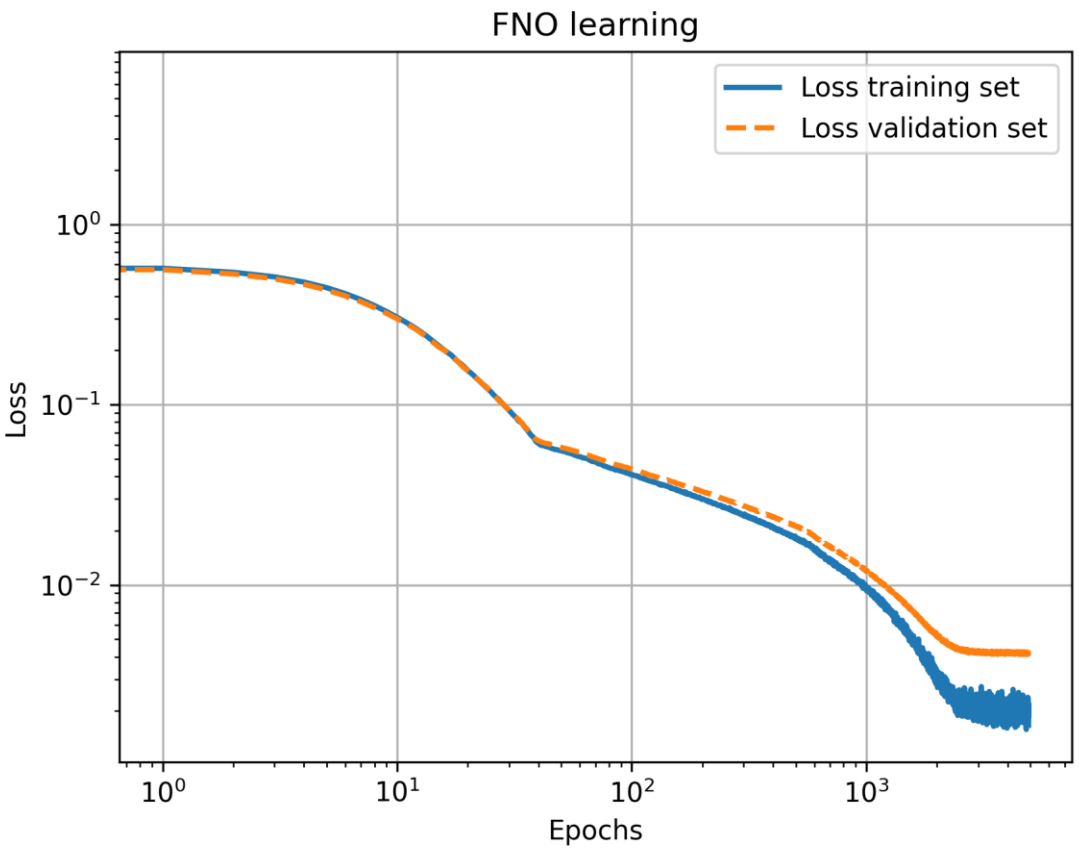
$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u^2}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \qquad (x,t) \in \Omega$$

$$u(x,0) = u_0 \qquad x \in (0,1)$$

We want to learn the map from the initial condition to the solution at the end time:

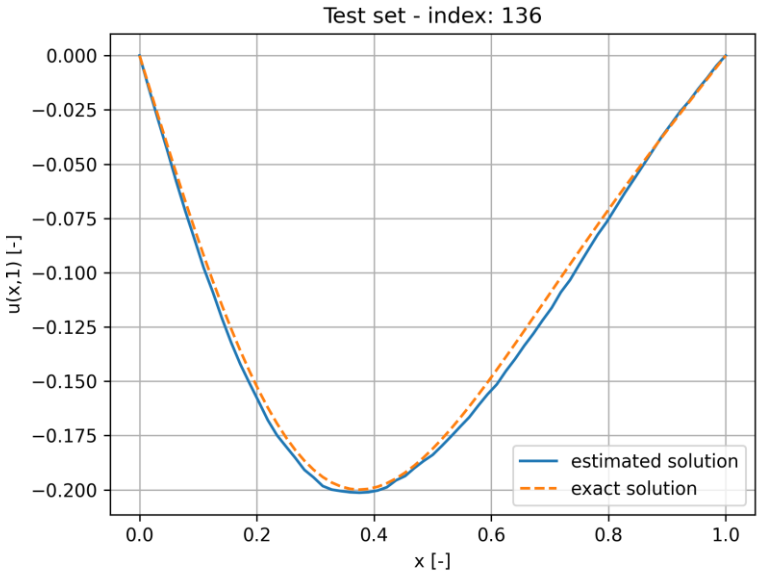
$$g^\dagger : L^2((0,1);\mathbb{R}) \to H^r((0,1);\mathbb{R}) \qquad u_0(x) \mapsto u(x,1) \qquad \forall r > 0$$

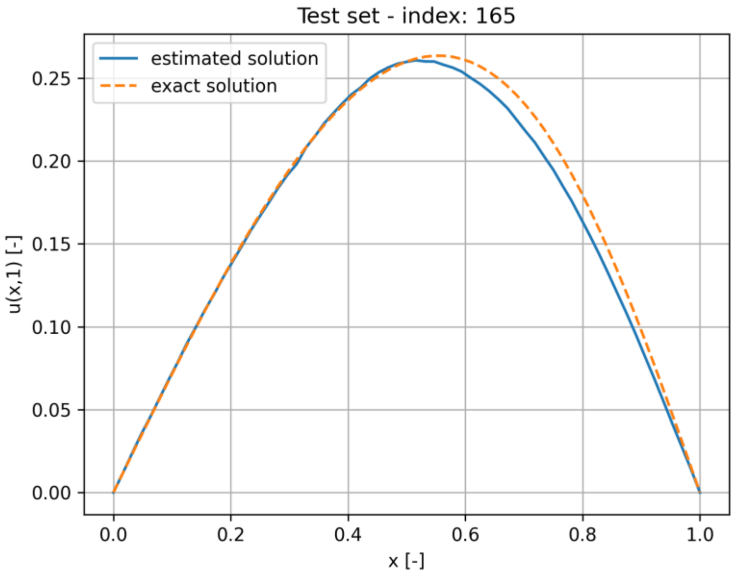
1D – Burgers equation

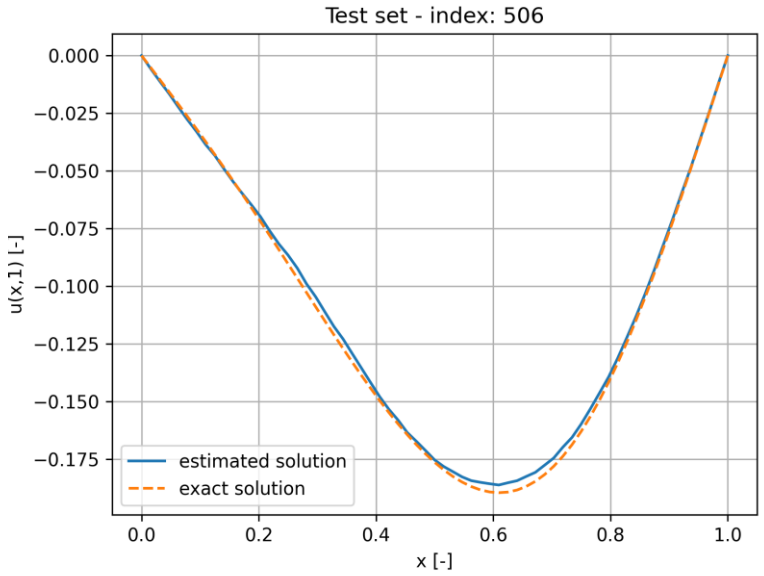


- We trained a FNO on a wide range of tuples of input/output generated by several u_0.
- The *exact* solution for training the FNO has been computed by a fine grid finite difference scheme.
- The trained FNO has:
  - 11 Fourier Layers
  - 7 modes in the FFT
  - 256 dimension in the Fourier space

1D – Burgers equation FNO performance







2D – Darcy equation

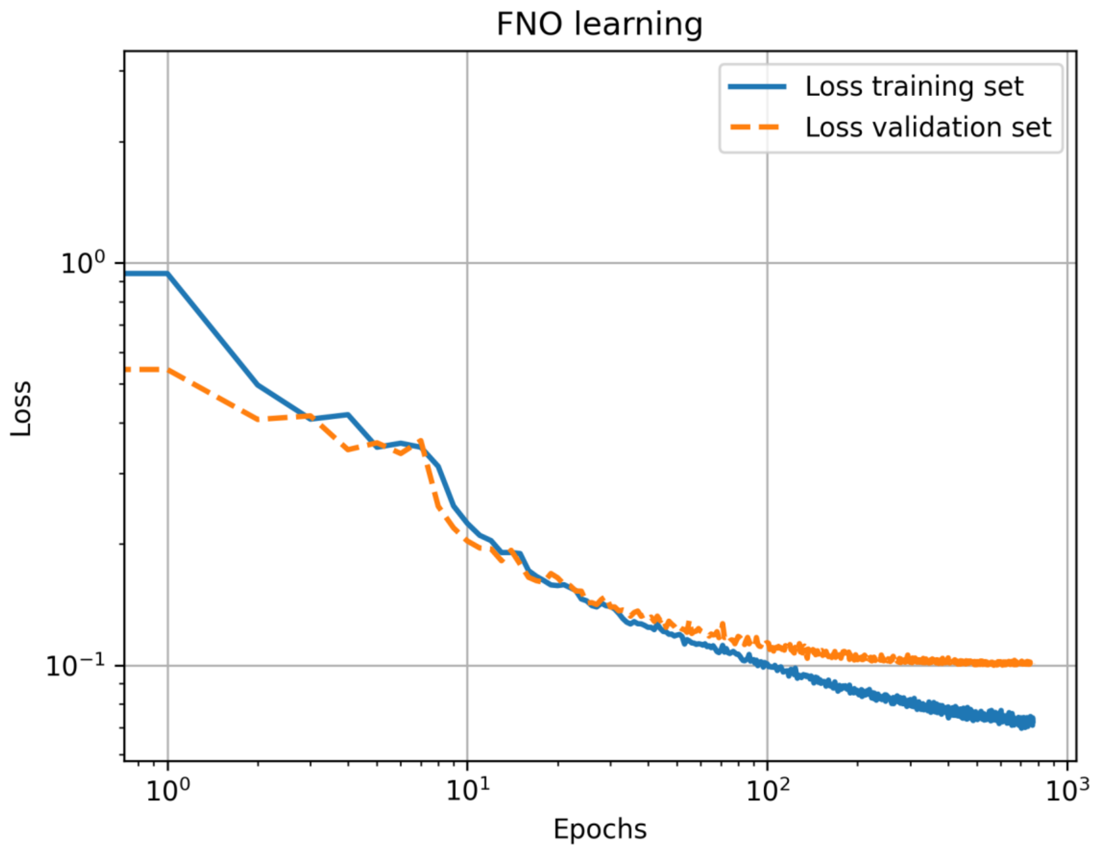
Consider the Darcy equation in $\Omega = (0,1)^2$ with $a \in L^\infty(\Omega; \mathbb{R})$ and $f \in L^2(\Omega; \mathbb{R})$

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

We want to learn the map from the diffusion coefficient a to the solution:

$$g^\dagger : L^\infty(\Omega; \mathbb{R}) \to H^1_0(\Omega; \mathbb{R}) \qquad a(x,y) \mapsto u(x,y)$$

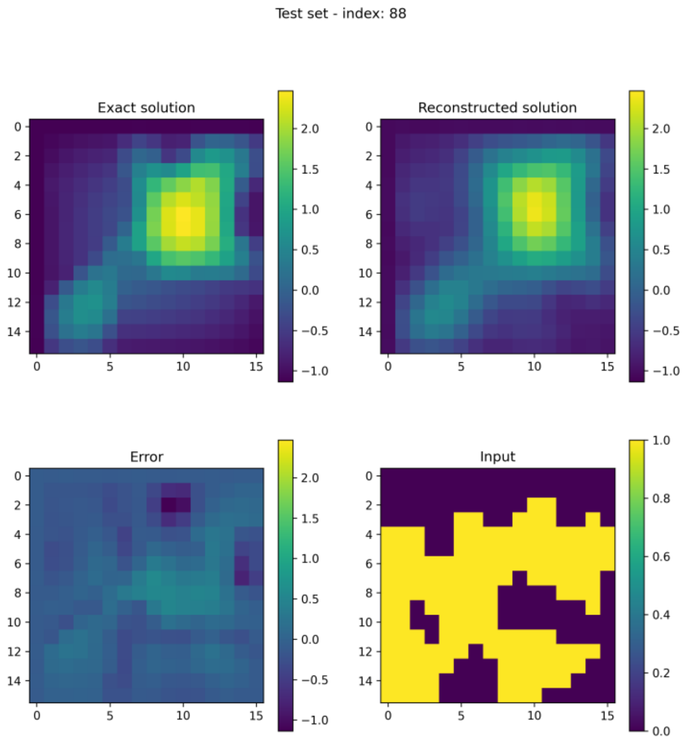
2D – Darcy equation

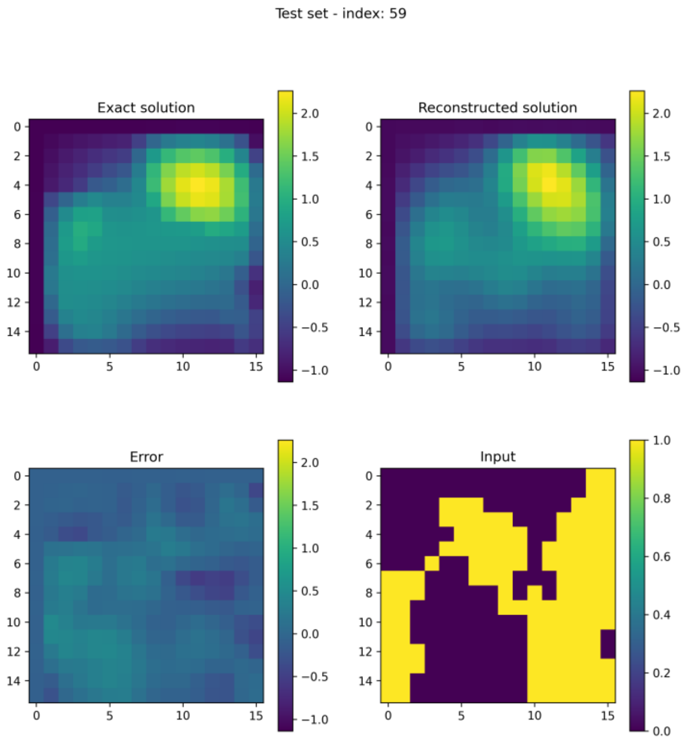


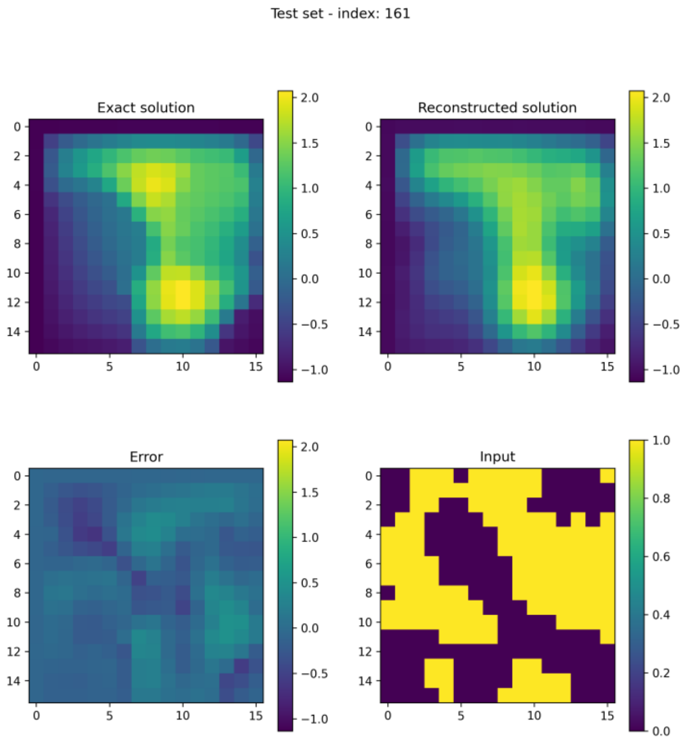
- We trained a FNO on a dataset found at https://github.com/NeuralOperator/neuraloperator.

- The trained FNO has:
  - 11 Fourier Layers
  - 7 modes in the FFT
  - 32^2 dimension in the Fourier space

2D – Darcy equation FNO performance







Coupled EnKF - FNO model

Burgers equation in time

Consider the Burgers equation in $\Omega = (0,1) \times (0,0.25]$ with $u_0 \in L^2(0,1;\mathbb{R})$ and $\nu = 0.025$

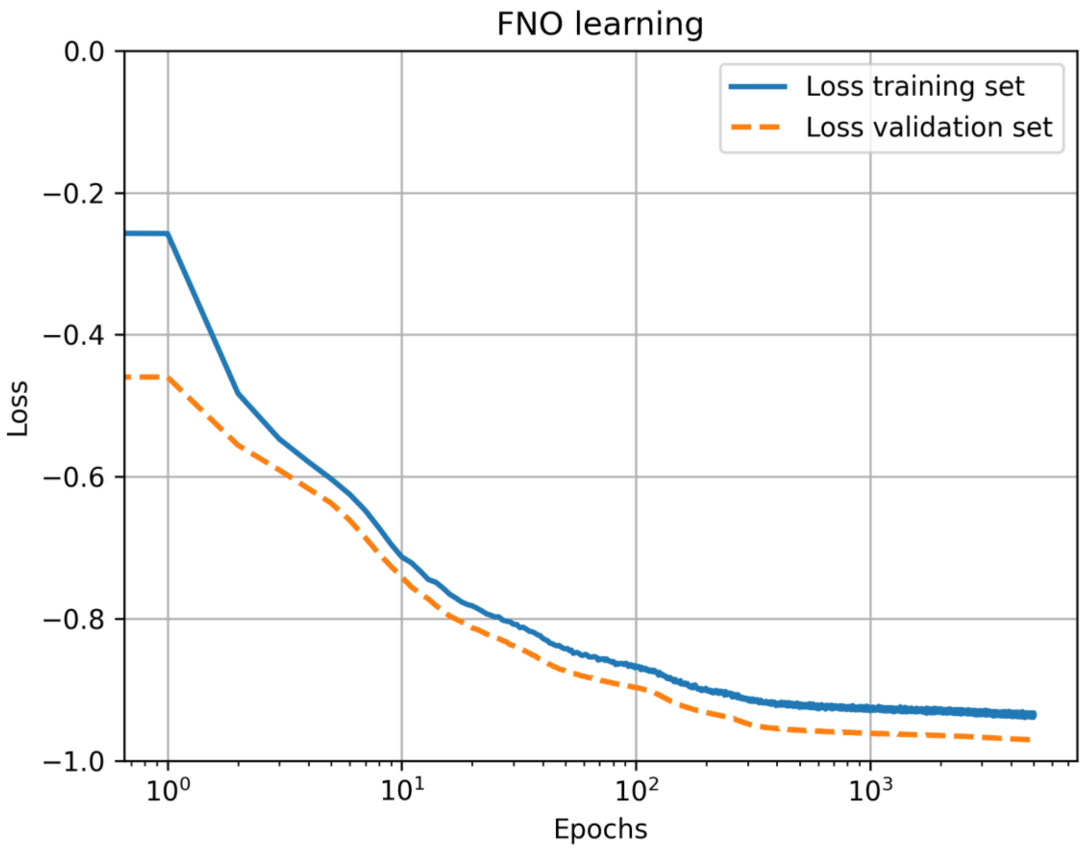
$$\frac{\partial u}{\partial t} + \frac{1}{2}\frac{\partial u^2}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2} \qquad (x,t) \in \Omega$$

$$u(x,0) = u_0 \qquad x \in (0,1)$$

We want to learn the map from the solution at time t to the solution at time $t + \tau$ where τ is the time discretisation $\tau = 1/64$:

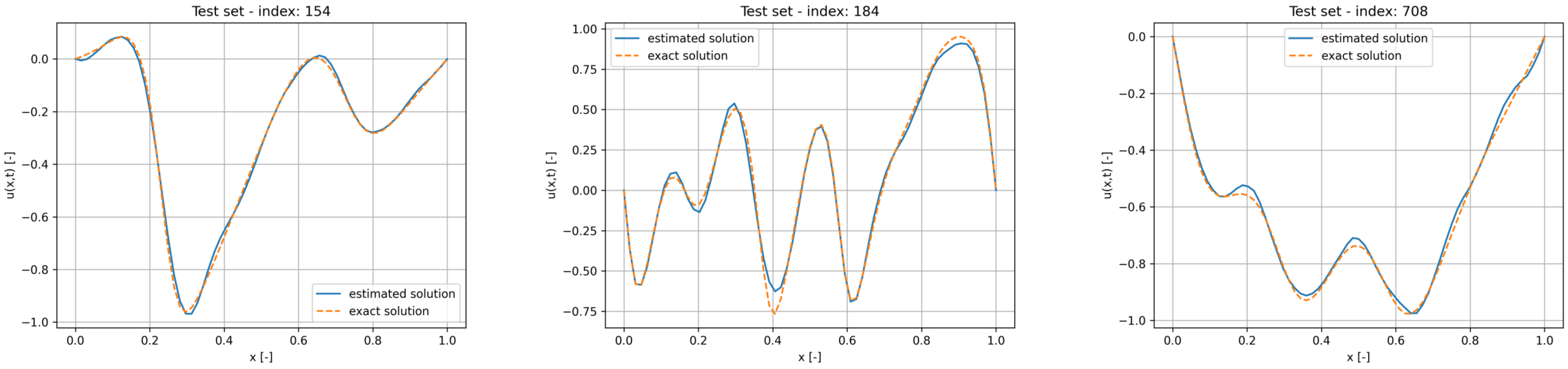
$$g^\dagger : H^r((0,1);\mathbb{R}) \to H^r((0,1);\mathbb{R}) \qquad u(x,t) \mapsto u(x,t+\tau) \qquad \forall r > 0$$

Burgers equation in time

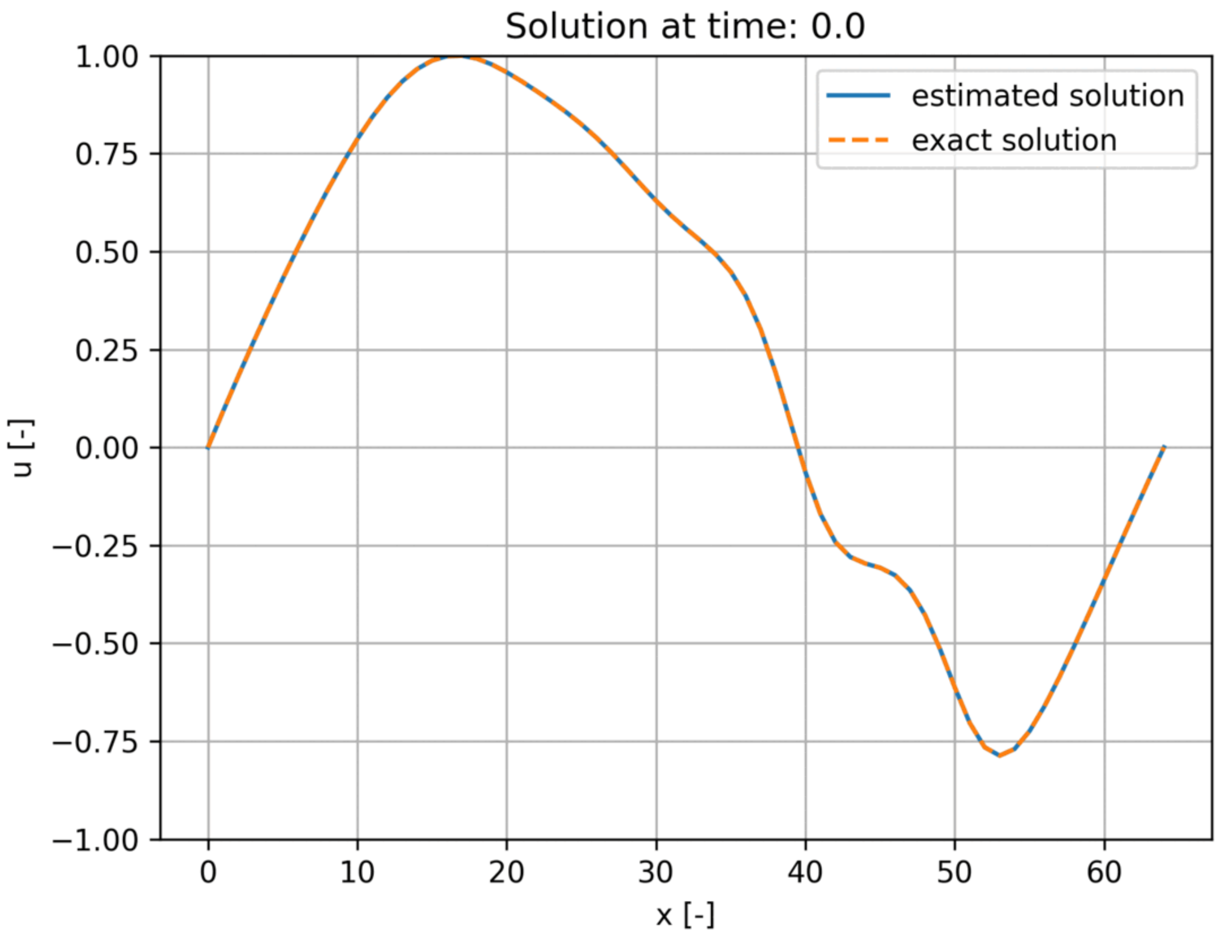


- We trained a FNO on a wide range of tuples of input/output generated by several u_0.
- The *exact* solution for training the FNO has been computed by a fine grid finite difference scheme.
- The trained FNO has:
  - 3 Fourier Layers
  - 14 modes in the FFT
  - 512 dimension in the Fourier space

Burgers equation in time – FNO performance



Burgers equation in time – EnKF model with FNO as transition function



Burgers equation in time with state-parameter

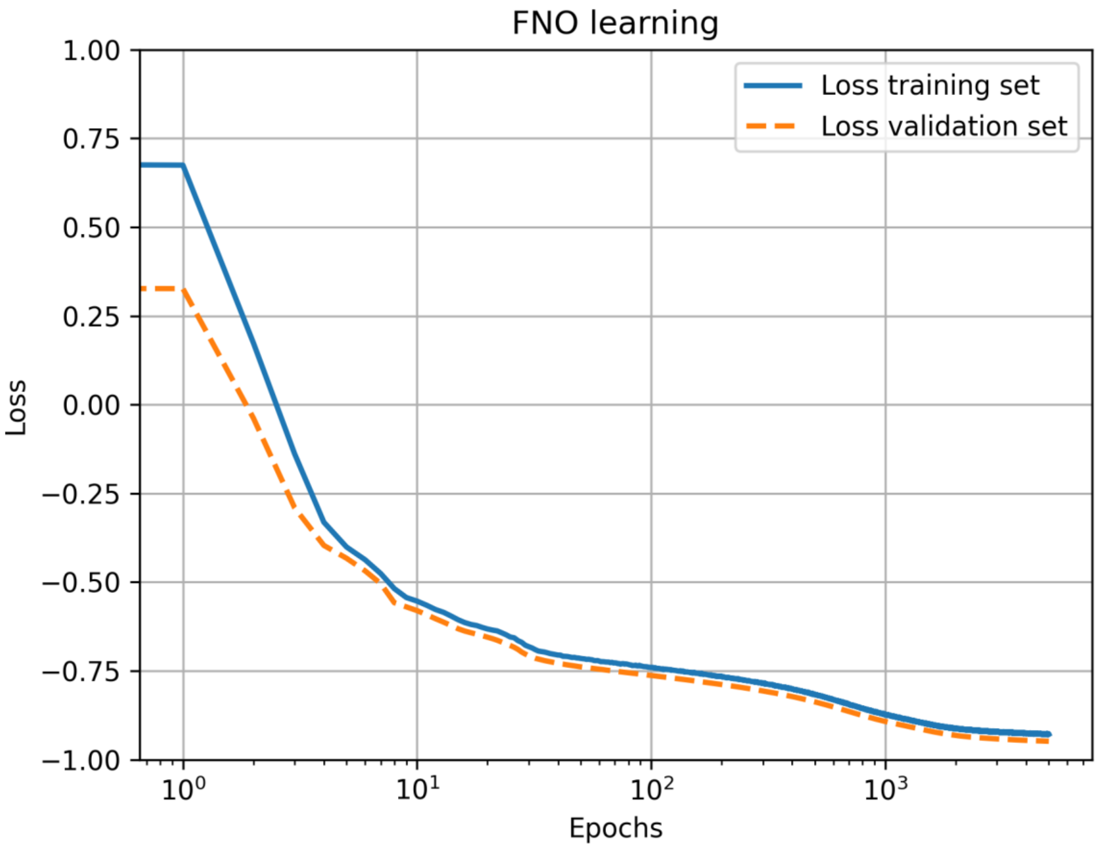
Consider the Burgers equation in $\Omega = (0,1) \times (0,0.25]$ with $u_0 \in L^2(0,1;\mathbb{R})$ and $\nu = 0.025$

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

We want to learn the map from the solution at time t to the solution at time $t + \tau$ where τ is the time discretisation $\tau = 1/64$:

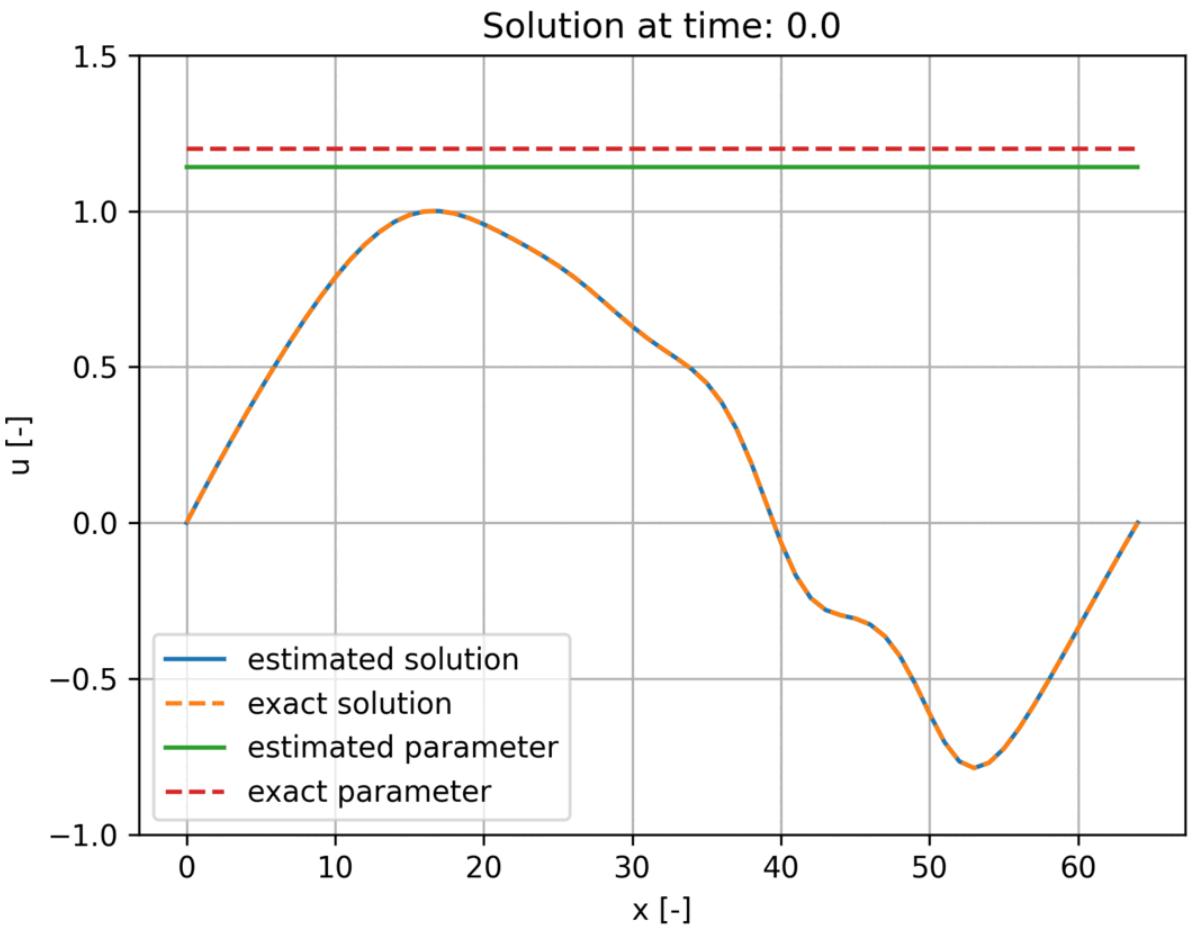
$$g^\dagger : H^r((0,1);\mathbb{R}) \times \mathbb{R} \to H^r((0,1);\mathbb{R}) \qquad (u(x,t),c) \mapsto u(x,t+\tau) \qquad \forall r > 0$$

Burgers equation in time with state-parameter – FNO performance



- We trained a FNO on a wide range of tuples of input/output generated by several u_0 and c.
- The *exact* solution for training the FNO has been computed by a fine grid finite difference scheme.
- The trained FNO has:
  - 3 Fourier Layers
  - 14 modes in the FFT
  - 512 dimension in the Fourier space

Burgers equation in time with state parameter – EnKF model with FNO as transition function



Thanks for the attention