POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Learning the effective dynamics of complex multiscale systems

Advanced Programming for Scientific Computing

Cattaneo Federico, Dotti Emanuela, Fedrizzi Daniele

Advisor: Stefano Pagani

Academic year:
2021-2022

Abstract: In the context of generating accurate predictions in multiscale systems, it is crucial to effectively capture the system dynamics. In this context, this report presents the implementation of the Learning Effective Dynamics (LED) model. The LED relies on surrogate models, employing an autoencoder and a Recurrent Neural Network. The code outlined in this report guides the user through the dataset generation, the construction and training of neural networks, giving also examples in the reproduction and evaluation of the LED. The entire framework is tested and evaluated through two case studies: the Van Der Pol oscillator and the Fitzhugh-Nagumo model.

Contents

1. Introduction

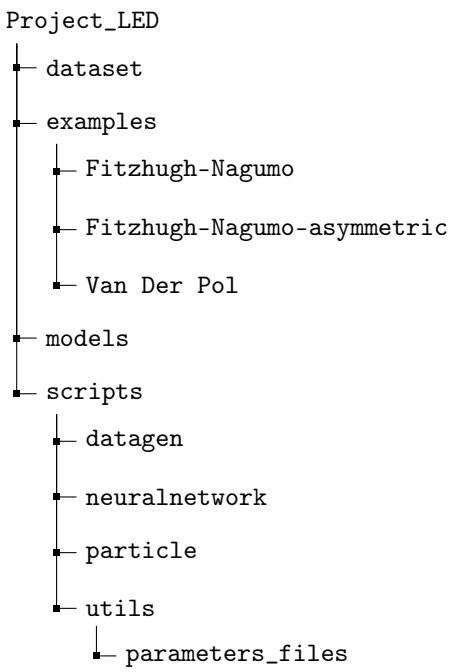
In the context of multiscale systems, it is crucial to effectively capture the system's dynamics and generate reliable predictions, as is the case in weather forecasting.
Large-scale simulations that encompass all system dynamics are often prohibitively expensive, and reduced-order models commonly rely on linearization, which are not effective in case of strongly nonlinear models.
For this reason, P.R. Vlachas and G. Arampatzis introduced the Learning Effective Dynamics (LED) framework in their paper [16].

In this context, our project aims at building a structure able to reproduce the LED framework as described in [16] and apply it to some concrete examples. The function of the network is explained in detail in Section 4.4.

The report is structured as follows: first, we introduce the structure of the code and some useful tools, like the abstract class ABC in python. Subsequently, we inspect the data generation through the two abstract classes DataGen and GenerateParticle, and dealing with the parameters handling. Section 4 presents the three classes used for the Neural Network: Autoencoder, RNN and LED.
In order to test the robustness, efficacy and user-friendliness of the entire structure, we use it on two applications: Van Der Pol oscillator and Fitzhugh-Nagumo models, that are presented in sections 6 and 7. An additional section reports the extension of the Fitzhugh-Nagumo model to the asymmetric autoencoder. Finally, section 9 reports some conclusion and future developments.

2. Code structure and Tools

The project's objective is to construct a framework capable of replicating the LED framework. Initially, we aim to empower the user to create a dataset for training the network as needed. Subsequently, we seek a structure that is adaptable, allowing the user to customize the network according to his/her requirements.
To achieve this flexibility and enhance code readability, we have chosen an object-oriented structure.
The code is accessible through the GitHub repository [5] and its structure is the following:



The folder `dataset` contains all the datasets used for the implementation of the two examples under study. It is also the default directory for all the new datasets created through the function `create_dataset`.
The folder `examples` contains the sub-folders for each example to which we apply the LED framework. Further details in the next section.
The folder `models` contains all the pre-trained networks we use for the examples. This is also the default directory when training and saving a new network.
The folders `datagen` and `particle` contain the scripts for the data generation.
The folder `neuralnetwork` contains the 3 classes Autoencoder, RNN and LED.
Finally, the folder `utils` contains the file `utils.py` with some useful functions used all over the code, and the file `params.py` containing the classes for the parameters handling (see section 3.3). The sub-folder `parameters_files` contains the files with the parameters for the solver and a README document that explains the layout.

2.1. How to run the code

To run the code and reproduce the results, navigate the `examples` folder. Inside there are 3 sub-folders.
In each sub-folder are some Jupyter notebooks, each one referred to the specific example of the folder.

The notebooks are the following:

- `main`
  By running this notebook, you can replicate the results concerning the LED. The file specifies the name of the pre-trained autoencoder and recurrent neural network, and employs the LED class to process the data and reproduce the LED framework.
  The user has just to run the entire notebook, or to change the name of the dataset, of the autoencoder or of the RNN to be uploaded.
  First, the object LED is initialized through the names of the pre-trained autoencoder and RNN, and the desired length in time of the prediction. Then, the method `run` allow to run the network according to the framework described in section 4.4. Finally, the method `compute_error` returns the errors of the prediction as explained at the end of the same section.
- `dataset_creation`
  This file is designed to guide the reproduction of dataset creation. It provides a step-by-step explanation of the procedure for generating a new dataset based on the model at hand.
  You can either run the notebook as it is or customized it according to your preferences.
  Since an ordinary differential equation will be solved, a solver object from the class GenerateParticle (refer to section 3.2) is instantiated.
  You can change the parameters of the solver by creating a new parameter file (refer to the README file inside `scripts/utils/parameters_files`) or by directly passing them to the constructor.
  Moreover, you can specify the dataset's name and parameters for the specific problem. Through the call to the function `create_dataset` (contained in `scripts/datagen`) you can define the number of samples to generate and the number of processors for the parallel execution. The input value `batch_size` allows

you to group samples, while the boolean variable `remove_samples` enables the elimination of samples after saving them in a single file. Adjusting `batch_size` is particularly useful for handling resource-intensive data generation, as in the case of the Fitzhugh-Nagumo model.
The notebook concludes by visualizing the generated dataset.

- `test_autoencoder`
  By running this notebook, you can test separately the autoencoder used in the LED framework (i.e. used in the `main` file). You just need to run the entire notebook.
  Through the creation of the Autoencoder object, by passing the model name to the constructor, the pre-trained autoencoder will be automatically uploaded.
- `test_rnn`
  As the previous function, the aim of this notebook is to test separately the RNN used in the `main` file for the LED class.
- `train_autoencoder`
  You can either run this file without modifying anything, or customize it.
  The purpose of this notebook is to give the possibility to the user to build and train his own autoencoder. For the customization of training and building parameters, please refer to Section 4.1.
  Once the object Autoencoder is initialized with the specified parameters (or the default ones), the notebook calls the method `get_data` to prepare the dataset for the training. Moreover, the methods `build_model` and `train_model` are called to respectively build and train the autoencoder accordingly to the specified parameters.
- `train_rnn`
  As the previous function, the objective of this notebook is to build and train a recurrent neural network. You can either run this file without modifying anything, or customize it. For customization please refer to Section 4.3.

If the user wants to create a new example, he has to create a new sub-folder inside the folder `examples`.
If he wants to create a dataset for the new example, it is necessary to build a new class that inherits from DataGen, since the generation of the dataset is problem-specific.
Furthermore, a new `if` instance with the name of the model has to be added inside the `factory.py` file (`scripts/datagen`).

Inside `scripts/particle` you can find also the file `test_particles.ipynb`. The purpose of this notebook is to test separately the solvers implemented inside the same folder.

2.2. ABC class

To address the problem at hand we employ an object-oriented approach, through the use of specific classes.
For the data generation process we use ODE solvers, that we deal with using the ABC class in Python ([1]). Given its attributes, we believe the most suitable approach involves the inheritance mechanism.

An ABC class is an Abstract Base Class. It is a kind of blueprint that cannot be instantiated on its own. Instead, it is meant to be sub-classed by other classes, which then provide implementations for its abstract methods. Abstract methods declared in the ABC must be implemented by any concrete subclass.

The ABC module allows us to create structured, well-defined class hierarchies by enforcing the implementation of certain methods in subclasses, ensuring better code organization and clarity in design.

2.3. Tensorflow

For the implementation of the networks we make use of the Python Library Tensorflow ([7]).
It is widely valued for building neural networks due to its versatility, efficiency, and extensive community support. Its flexible architecture allows users to design and train a variety of neural network models, while its optimized performance ensures efficient computation, making it suitable for our purposes.
By employing this library, users have the flexibility to specify the number of Convolutional and Dense layers, determine the number of units for each layer, select the optimizer, and adjust other hyperparameters crucial for constructing and training a neural network that fits its purpose. This ensures a customized network design while preserving the user-friendly nature of the code.

2.4. Constructor overloading

Generating data requires the user to input a large number of parameters, therefore we add the option to import them from an external file. However, Python doesn't normally support constructor overloading, which lead us

to the use of decorators to navigate around this issue.
In the sub-classes of GenerateParticle, we use the `singledispatchmethod` decorator. Thanks to it, we are able to alter the behaviour of `__init__` depending on the typing of its first non-self argument. Using this tool, we decorate so that if the user calls it by passing the parameters as a string representing the path, the call will be redirected to a function handling file reading.
This is only possible because in both cases the constructor receives as input a parameter list and a function. However, when instantiating a SovlerParams object, the number of required arguments is different. To account for this, we apply the classmethod decorator to the `get_from_file` method, which allows the user to explicitly call it as an alternative to the `__init__` constructor.

In the file `test_particles.ipynb`, you can see an example of the constructor overloading for a ThetaMethod object.

3. Data Generation

In order to be able to train a neural network for complex multiscale systems, we need a large dataset. Since we have no such dataset at our disposal, we build a class able to solve ordinary differential equations with different solvers. With such a class, we are then able to generate datasets for specific problems, like the ones we deal with in sections 6 and 7.
The implementation of the classes for the data generation is contained inside the .py files of the folder `scripts/datagen`.

3.1. DataGen

DataGen is an ABC python class. Its role is to supply a structure to generate and save a dataset.
Its structure is the following:

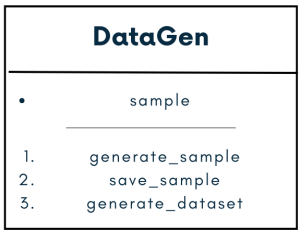


Figure 1: DataGen class structure with members and methods.

The method `generate_sample` is an abstract method, and it is subscribed from each sub-class that is able to generate the sample according to a specific model. In our case, we have the sub-classes VanDerPol and FitzhughNagumo that inherit from DataGen and are able to create a dataset according to the specific model at hand.
The second method is called by the sub-classes to save a sample.
Finally, the method `generate_dataset` takes as argument the number of samples and the number of processors for the generation of multiple samples. It can also take the value x_0, which in our case is the starting point for the simulation of the models.
These samples are generated through parallelization, employing the Pool class by the Multiprocessing library. Further details in the next section.

3.1.1. Time and memory

While creating the dataset for the Fitzhugh-Nagumo model, we encountered problems with the high computational cost and the large memory space needed.
To overcome these two issues, we make use of parallelization and compression.
The latter is possible though the method `save_sample` that allows to save the sample in the `.npz` format.
The parallelization instead is possible through the library `multiprocessing` ([3]) and its class `Pool`.
The Pool Class automatically allocates tasks to the available processors in a First-In-First-Out (FIFO) scheduling manner. By employing its method `starmap`, we can concurrently execute the function `generate_sample` on various input values (the name with which the sample is saved and the starting point x_0). Each input value is assigned to a separate process, enabling parallel execution.

The first attempt of the dataset creation through serial code took more then 30 hours. With the parallel code, we've managed to more than halve the total computational time.
Figure 2 shows the computational time with respect to the different number of processors used.
As we can see there is a substantial gain when employing the parallel strategy.

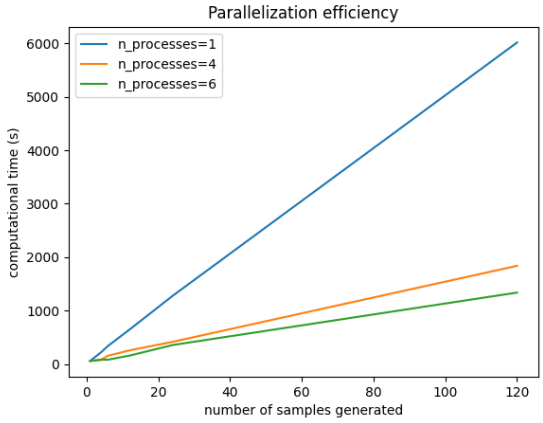


Figure 2: Computation time comparison between 1, 4 and 6 processes.

3.1.2. Save and store

As the primary data structure employed to store the sample, we opt for a NumPy array ([4]). This choice stems from our prior knowledge of the sample's dimension. Since the time step size dt and the final time T are predetermined by the user, we possess a priori information about the sample's size. Additionally, as we do not require key-based access to the sample, the NumPy array serves our purpose effectively. These considerations are applicable not only to the DataGen structure but also to the GenerateParticle structure, as detailed in section 3.2. Considering these specific requirements, the numpy array structure emerges as an optimal solution in terms of both memory efficiency and computational performance.

3.2. GenerateParticle

The GenerateParticle class is an Abstract Class that aims to be a support for solving ODEs of the type:

$$\begin{cases} \mathbf{u}'(t) = f(t, \mathbf{u}) \\ \mathbf{u}(t_0) = \mathbf{u}_0 \end{cases} \tag{3.1}$$

Its abstract method is `generate`. This method is implemented by the sub-classes that aim at solving ODEs with a specific solver.
The structure of the class is the following:

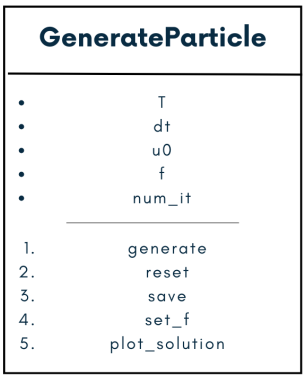


Figure 3: GenerateParticle class structure with members and methods.

The function `set_f` allows to set the function f of the problem. This is particularly useful when generating

some samples that necessitate alterations to the forcing term, as exemplified in the implementation of the Fitzhugh-Nagumo model.
This method invokes the function `to_numpy` from the `utils.py` file that permits the conversion of the function f into a NumPy array, as mandated by the code.
The sub-classes of GenerateParticle are: Multi-steps, Runge Kutta and Theta method. They provide the implementation of the numerical method to solve the ODEs through the method `generate`, that overrides the abstract one of the base class.
The final inheritance structure for the class GenerateParticle is shown in Figure 4.

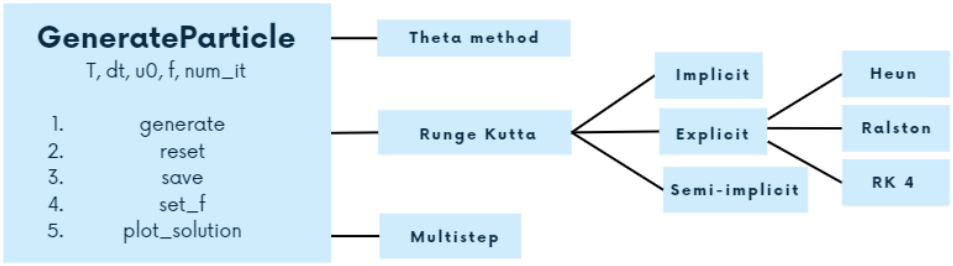


Figure 4: ABC GenerateParticle: inheritance scheme.

RungeKutta is a sub-class of GenerateParticle and at the same moment is itself a base class for each specific Runge Kutta method.
The s-stage Runge Kutta method to solve an ordinary differential equation is divided into explicit and implicit methods. They both rely on the following formula, taken from [12]:

$$u_{n+1} = u_n + h\sum_{i=1}^{s} b_i k_i, \quad n = 1, ..., T-1 \tag{3.2}$$

where in the case of explicit methods:

$$\begin{aligned} k_1 &= f(t_n, u_n), \\ k_2 &= f(t_n + c_2 h, u_n + (a_{21}k_1)h), \\ &... \\ k_s &= f(t_n + c_s h, u_n + (a_{s1}k_1 + a_{s2}k_2 + a_{s,s-1}k_{s-1})h). \end{aligned}$$

For what concern the implicit methods, the coefficient k_i has the following definition:

$$k_i = f(t_n + c_i h, u_n + h\sum_{j=1}^{s} a_{ij}k_j) \quad i = 1, ..., s.$$

In any case, the Runge Kutta method is based on the definition of the arrays A, b and c, that contain the coefficients from equation (3.2). Together they constitute the Butcher tableau as represented in figure 5.

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} = \begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b^T} \end{array}$$

Figure 5: Runge Kutta Butcher array.

Since these arrays have specific mathematical constraints, the constructor of the base class Runge Kutta calls a function of the python file `utils.py` that check whether the mathematical constraints are respected. If not, `assert` command is called and the function brokes.
Each sub-class then recalls the constructor of the base class through `super().__init__`. In this way the mathematical constraints for the Butcher tableau are checked even when instantiating a sub-class, because the constructor of the base class is always called. Plus, each sub-class calls, if necessary, specific controls on the arrays A, b or c, according to what is required by the specific method.

Additionally, for the case of an explicit method, an assert is called if the Butcher array given is implicit, but explicit method was called by the user.

Some specific methods are provided: Heun, Ralston and Runge Kutta 4. This way the user can relies on that implemented methods without inserting manually a specific Butcher tableau.

The Multistep class contains the Adams-Bashforth method with its constructor and the concrete method `generate`.
The s-steps Multistep method is defined from [10] through the formula:

$$\sum_{j=0}^{s} a_j u_{n+j} = h \sum_{j=0}^{s} b_j f(t_{n+j}, u_{n+j}), \tag{3.3}$$

with $a_s = 1$.
The Adams-Bashforth methods have coefficients $a_{s-1} = -1$ and $a_{s-2} = ... = ... = a_0 = 0$, with s being the order. Further improvement can include the implementation of additional specific sub-classes.

The ThetaMethod class instead takes as input a tolerance and the value of theta and implements the Theta Method for the solution of an ODE.
From [9] the equation reads:

$$\mathbf{u}_{n+1} = \mathbf{u}_n + h[\theta \mathbf{f}(t_n, \mathbf{u}_n) + (1 - \theta \mathbf{f}(t_{n+1}, \mathbf{u}_{n+1}))], \quad n = 1, ..., T-1 \tag{3.4}$$

You can test all the solvers inside `scripts/particle/test_particles.ipynb`.
In alternative, you can test the GenerateParticle class when reproducing the creation of the dataset of the examples through the Jupyter notebook `dataset_creation`, as explained in section 2.1.

3.3. Parameters Handling

In order to handle the parameters for the solvers, we build a customized class: SolverParams. The structure is outlined in the figure below.

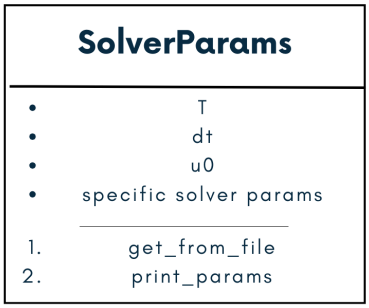


Figure 6: SolverParams class structure with its members and methods.

The constructor receives the final time T of the simulation, the time step size dt and the initial solution $u0$ as input.
Additionally, the parameters for the specific solver (e.g. θ for the Theta Method) must be provided.

It is possible to instantiate an object of this class by passing a file containing the necessary parameters.
Since directly overwriting the constructor `__init__` isn't feasible, the class includes the method `get_from_file`. This method is an alternative constructor that allows creating an instance of the SolverParams class by reading parameters from a file. It takes as input the class itself and the directory of the file to be read.
This method is called by the `singledispatch` method to initialize an object of type GenerateParticle through passing the directory of the file instead of the SolverParams object itself, as explained in Section 2.4.

To create a file for parameters handling, go inside `scripts/utils/parameters_files`. Here you can create a new `.txt` file following the guidelines contained inside `READMEparams.txt`. The usage of this class is shown inside the Jupyter notebook named `dataset_creation`, which is located inside the examples folder. In this context, the method `get_from_file` is used to create a SolverParams object for the solver responsible for solving the ODE pertinent to the specific example.
The class implementation is contained in `scripts/utils/params.py`. Here you can find also the parameter class to handle the model parameters for the Fitzhugh-Nagumo model.

4. Neural Networks

In order to build and train the LED model shown in Figure 10, we handle the structure with 3 different classes: Autoencoder, RNN and LED.
Each of these is built for the user to have deep control of the newtork (possibility of changing activation function, dense layer etc.) and easy-usage at the same time (default values and pre-built structure). All the details are shown in the following sections.
You can find the `.py` files containing these classes inside the folder `scripts/neuralnetwork`.

4.1. Autoencoder

The structure of the Autoencoder class is the following:

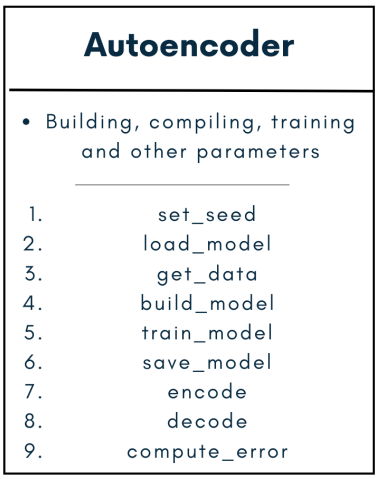


Figure 7: Autoencoder class structure with its methods.

The user is required to provide details regarding the latent space dimension.
Moreover, there's the option for the user to input additional specifications, encompassing the construction parameters (like layer specifications), compilation parameters (such as loss function and optimizer), and training parameters (including batch size and number of epochs).
It's important to note that these specifications, except the latent dimension, are pre-set with default values. This design choice aims to strike a balance between facilitating the ease of use of the class and granting users the flexibility to customize the neural network to their preferences.

For this last objective, the user can provide some specifications for the Dense and Convolutional Layers.
The notation is the following:

`dense` = [64, 128],
`conv` = [(8,3),(16,3),(32,3)],

where for the Dense Layer the numbers represent the units for each Dense in the network. In the Convolutional Layer, each tuple refers to a specific Convolutional Layer. The first value in the tuple indicates the number of filters, while the second value represents the kernel size.
The number of layers is not fixed: adding parameters to the inputs dense and conv, the user can choose the number of Dense and Convolutional layers.
The class Autoencoder provides some methods:

- `set_seed` sets the seed for both NumPy and Tensorflow libraries, for result reproducibility.
- `load_model` takes as input the name of a model, located in the `models` folder and loads that model's structure and weights.
- `get_data` takes the path to the data and upload it into the specific class member, according to the file extension. The supported extensions are `.npy`, `.npz` and `.csv`. The usage of this function is necessary to build and train the network using the `build_model` and `train_model` methods.
- `build_model` set the input shape according to the uploaded data and cycle over the dense and conv inputs to build the autoencoder. It takes in input an optional variable, `summary`. If `True`, the method prints the summary of the network.
- `train_model` calls the Keras (from the library [2]) `fit` function to train the autoencoder.

- `save_model` enable to save the autoencoder once trained. It is saved into `.json` and `.h5` formats.

The class also includes the methods `encode` and `decode`, allowing the separate utilization of the autoencoder's encode and decode functions on a dataset. These methods return the encoded or decoded dataset and accept the `save` variable as input, which defaults to `False`, enabling the optional saving of the dataset.
The `encode` method also accepts an optional input `smooth`. When set to `True`, this smoothens the dataset. We introduced this option based on our specific application needs.

4.2. Asymmetric autoencoder

In our study of the problems at hand we notice that, since the direct simulations of data are expensive in terms of both time and memory, an user might only be able to generate numerical data on a coarser grid. In order to solve this issue we decided to improve on our LED model so that it may be able to produce an output that is more refined than the original input.
The tool that allows us to perform this task is a new autoencoder, replacing the original one, characterized by an extra layer in the decoder that performs an additional upsampling; due to the difference between its input an its output, we named the class Autoencoder_asymmetric.
Regarding its implementation, this class derives from Autoencoder, overloading the following methods:

- `get_data`: the user is required to pass the output data via its path; after that, the input data will be computed by extracting, for each sample and for each instant in time, every other sample in the grid. This will result in an input data having halved grid sides, rounded up.
- `train_model`: the call to the `fit` function now receives as input data the coarser grid.

It should be noted that the method `build_model` has not been overwritten since the base class already accounts for the extra layer by performing a check on the shapes of the input and output.

4.3. Recurrent Neural Network

The RNN is composed of LSTM layers followed by Dense layers. LSTM is short for Long Short-Term Memory [14], and this kind of neuron is how the network, given a time series $\{s(t_i)\}_{i=0}^{n}$ in order to predict $s(t_{n+1})$, can "remember" what happened in instants t_{n-1}, t_{n-2}, ... and use them to forecast the following instant. Figure 8 shows how does the training of a RNN works. The dense layers are used then to put together what is extracted from the LSTM layers and make a proper prediction.

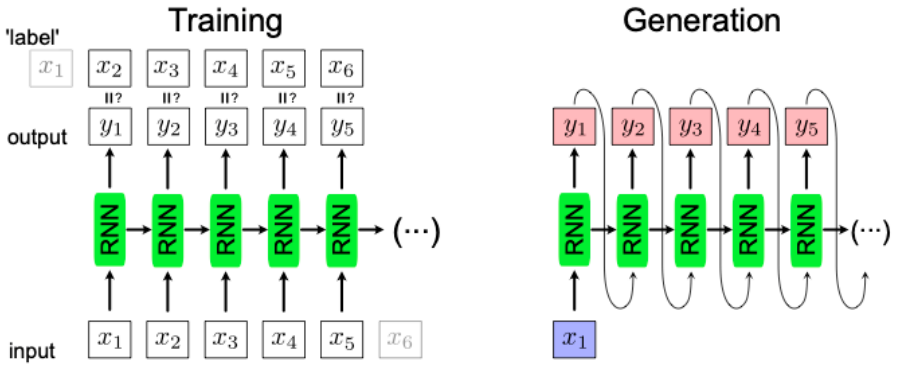


Figure 8: Diagram of how training works in an RNN. [6]

To handle this kind of network, the class RNN is structured as follow:

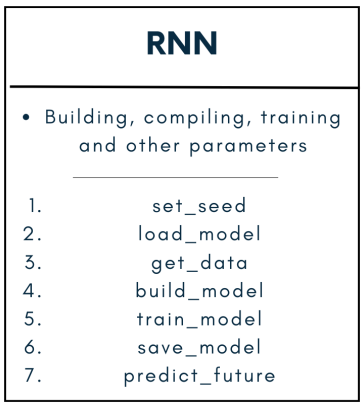


Figure 9: RNN class structure with its methods.

The methods of the class are basically the same as for the Autoencoder class, but suited for the case of a Recurrent Neural Network.

Also in this case the user can specify additional requirements for the layers:

`lstm` = [64, 128, 256],
`dense` = [256, 128, 64],

where for both the Dense and the LSTM Layers the numbers represent the units.
By using the input variable `bidirectional`, users have the option to specify whether they desire to construct a network with bidirectional LSTM layers.
Within the RNN class, the function `get_data` constructs sequences for the training set, aligning with the requirements of Recurrent Neural Networks, using the specified window size.
With an initial sequence as input, the function `predict_future` can forecast n future steps, where n is determined by the input variable `length`.

4.4. LED

The class LED handles the evolution of a dynamical model according to the structure presented in [16].
The LED model works as shown in figure 10.

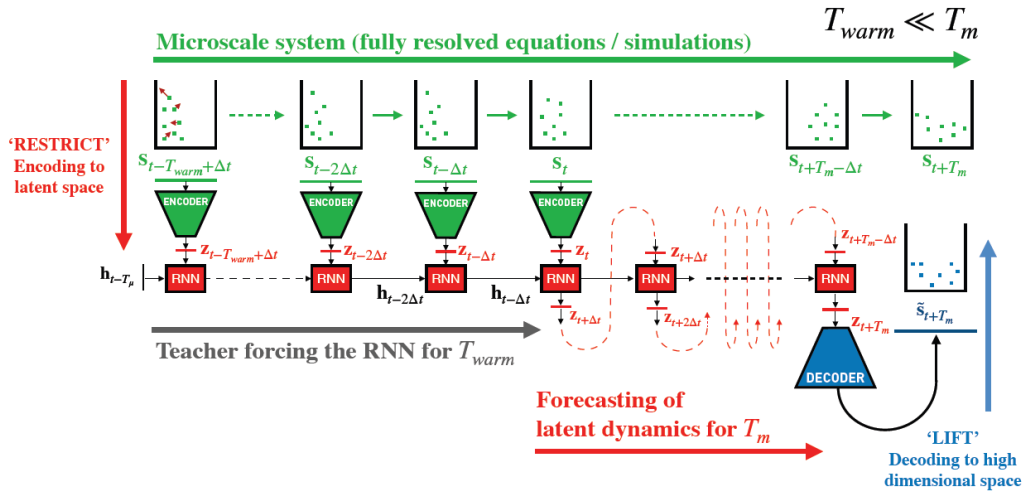


Figure 10: Scheme of the LED. [16]

The simulation involves high-dimensional dynamics represented by the state, $\mathcal{S}_t$, for a brief duration of T_{warm} time units. This time T_{warm} will be the length of the window size of the RNN. During this initial warm-up period, the state $\mathcal{S}_t$ undergoes encoding via an encoder network. The resulting outputs, denoted as z_t, are then recurrently fed into an RNN, sequentially updating its hidden state h_t. This process captures non-Markovian

effects within the system. Starting from the final latent state z_t, the RNN iteratively advances the lower-dimensional latent dynamics for a total period of T_m time units, where $T_m \gg T_{warm}$. In our code T_m is the input variable `length_prediction` specified by the user. Subsequently, the LED decoder transforms the last latent state z_{t+T_m} back to a high-dimensional representation $\mathcal{S}_{t+T_m}$. Through LED, traversing the lower-dimensional space is significantly more computationally efficient compared to evolving the high-dimensional system from fundamental principles. Consequently, LED accelerates simulations, enabling the exploration of dynamics across much longer time scales. It unveils regions within the state space that would otherwise be too complex or challenging to analyze.

The LED class receives input data from the microscale system and performs a complete process, encompassing encoding, forecasting within the latent space, and decoding back to the microscale dimension.

The structure of the class is the following:

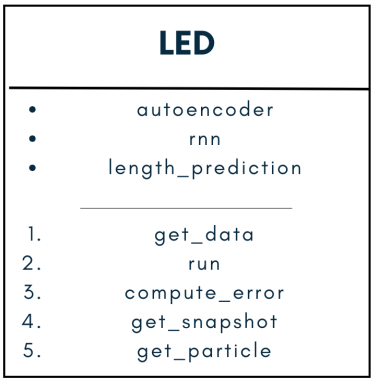


Figure 11: LED class structure with its members and methods.

The class takes in input the name of the pre-trained autoencoder and RNN models, together with the number of timesteps the user wants to predict.
The primary function, `run`, manages the prediction process along with all its associated steps, as depicted in Figure 10.
The function `get_snapshot` takes one or more time steps as input. It outputs, and optionally plot, the resulting dynamical system in a fixed time step (or steps).
The function `get_particle` recalls the previous function to output, and optionally plot, the trend over time of a single particle.
These two functions are specifically tailored for the Fitzhugh-Nagumo test case.
Given a test dataset, the function `compute_error` measures the network performance by computing the following errors:

- `Snapshot error`: for every snapshot in time, compute the Frobenius norm over the whole grid for each component, then compute the error matrix as the square root of their sum of squares
- `Particle error`: for every particle in the grid, compute the L^2 norm of each component, then compute the error array as their quadratic average.
- `Model error`: compute the L^2 norm of the Snapshot errors over the time steps.

The above-described errors are valid only in the context in which we use our network on grid-like data, such as the Fitzhugh-Nagumo case. In the case in which the data is a single time sequence, the errors are redefined as follows:

- `Snapshot error`: L^2 norm of the difference function, at fixed times.
- `Particle error`: first the L^2 norm of each component is computed, then the particle errors the square root as their sum
- `Model error`: the same as the particle error

5. Applications

5.1. Classes for case studies

Each specific case study is managed through a class that inherits from the base class DataGen.
This class will contain the function `generate_sample` that generates the sample according to the specific model under study, overwriting the abstract method of the base class DataGen.

Following this approach, we have created two classes, VanDerPol and FitzhughNagumo, that inherit from DataGen and have the following structure:

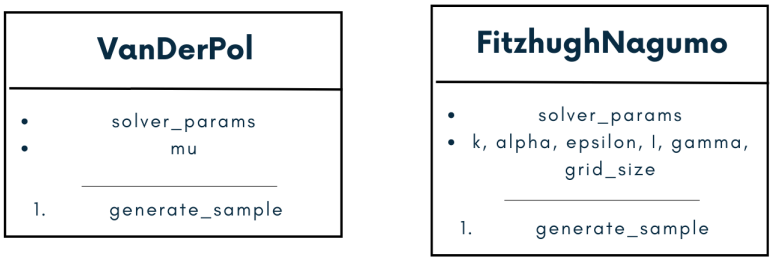


Figure 12: VanDerPol and FitzhughNagumo classes structure with their members and methods.

5.2. Object factory

In order to allow the user to more easily integrate new case studies without the need to alter the more elaborate scripts, we opted to use an object factory. It is designed as a function, receiving as input the name of the solver and its parameters and returning the corresponding object.
If a user needs to create a new test case, it will simply need to create a class deriving from Datagen and add it to the `Model_Factory` function in the `factory.py` file.

6. Case Study: Van Der Pol

6.1. Governing equations and dataset

For the application of the implemented framework, we begin with the ODE provided by the Van Der Pol oscillator [8].
We proceed with the implementation of a single Recurrent Neural Network without employing the autoencoder.
The Van Der Pol oscillator is a non-linear damped oscillator governed by the following equations:

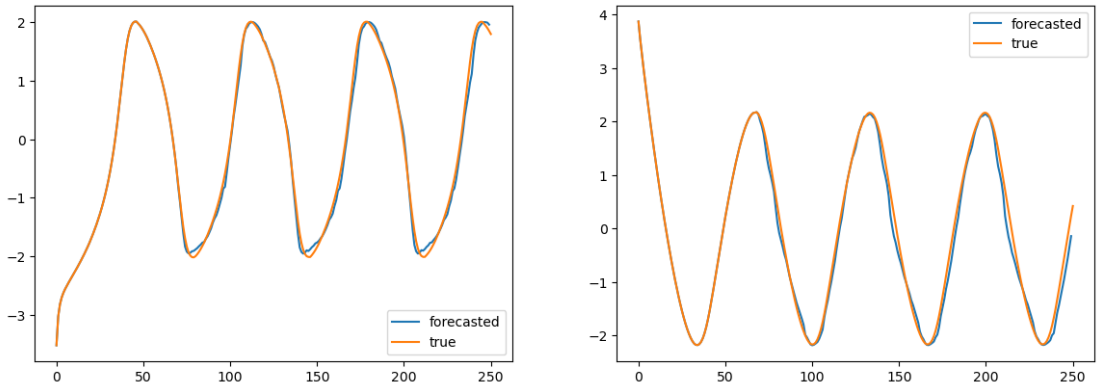
$$\begin{cases} x' = \mu(x - \frac{1}{3}x^3 - y) \\ y' = \frac{1}{\mu}x, \end{cases} \tag{6.1}$$

where in our case μ is fixed equal to 1.

The dataset is obtained by solving the ODE with the Theta Method, starting from random points in between $[-5, 5]$x$[-5, 5]$.
The parameters for the solver are contained inside `scripts/utils/parameters_files/parameters_VDP.txt` and can be modified following the instruction contained in the `READMEparams.txt` file.
As explained in detail in Section 2.1, the creation of the dataset can be entirely reproduced by running the jupyter notebook `dataset_creation` inside the folder `examples/Van Der Pol`.

6.2. Recurrent Neural Network and LED

The RNN is build and trained using all the default parameters, with the exception of the window size, which is set to a fixed value of 70 (approximately one-quarter of the 250 time steps in the problem).
The pre-trained RNN produces the result depicted in Figure 13, showcasing a comparison between the predicted signal and the original one.
The same results can be obtained employing the class LED.
To reproduce the results, you can either run the file `main.ipynb` or `test_rnn.ipynb` inside the folder `examples/Van Der Pol`.



(a) x-component. (b) y-component.

Figure 13: RNN results on the test set for the Van Der Pol oscillator.

7. Case Study: Fitzhugh-Nagumo model

The second example is the application to an Action Potential model.
The section is structured as follows. First, the physical governing equations are explained. Then, the structure of the dynamical model under examination is shown, based on the physical model. Subsection 7.3 shows the details about the data created for the simulation. Then, the autoencoder and the RNN are presented. Last, the results of the LED are shown.

7.1. Governing equations

We want to test the network on a simple dynamical system based on the Fitzhugh-Nagumo equation [15].
This model is one of the simplest for the action potential that governs the depolarization and repolarization of a membrane potential (difference of potential between the interior and exterior of a cell).
The equation reads:

$$\begin{cases} u'(t) = Ku(t)(u(t) - \alpha)(1 - u(t)) - w(t) + I_{app} \quad t \in (t, T] \\ w'(t) = \epsilon(u(t) - \gamma w(t)) \end{cases} \tag{7.1}$$

where $u(t)$ is the adimensionalized membrane potential, $w(t)$ is the adimensionalized recovery variable that controls the repolarization process and I_{app} is the stimulus. This stimulus is defined as:

$$I_{app} = \mathcal{I}_{app} \mathbb{1}_{[t_{begin}, t_{end}]}(t), \tag{7.2}$$

with $\mathcal{I}_{app}$ being the intensity of the stimulus, equal to 0.125 in our case.
All the other variables are constants that need to be chosen. In our problem-specific case they are set as $K = 8$, $\alpha = 0.15$, $\epsilon = 0.01$ and $\gamma = 0.1$.

7.2. Dynamical model

The dynamical system under investigation lies on a unit square. The `grid_size` parameter defines the points in which the signal is evaluated. The stimulus departs from a firing point x_0 and it propagates through the square according to a fixed velocity v.
Each point of the grid uses the class GenerateParticle to solve the ODE system (7.1). This mechanism is shown in figure 14. The right-hand side of the figure report the plot of the time-varying solution of system (7.1) in a single point of the grid.

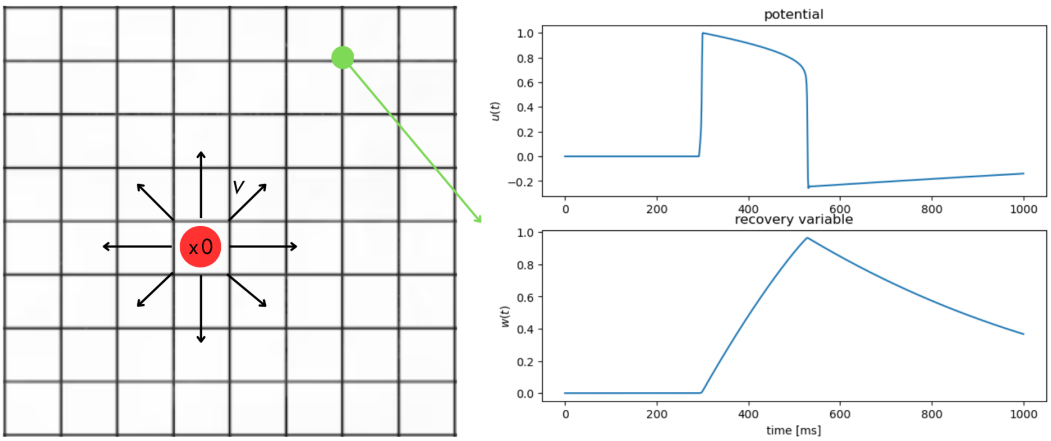


Figure 14: Working principle of the dynamical model being investigated.

This propagation is implemented through the method `generate_sample` of the class FitzhughNagumo. This class is problem-specific and it inherits from the base class DataGen.

To reproduce the generation of this kind of dataset, you can run the Jupyter notebook `examples/Fitzhugh-Nagumo/dataset_creation`. This notebook will automatically create a dataset made of 1 sample, generated from a grid of dimensions 20x20. From the mentioned notebook, you can also change the number of samples, the number of processors for the parallel execution, and the grid dimension.

7.3. Sampling

In order to train the networks, a substantial dataset is required.
Our dataset consists of 100 samples, each containing a 20x20 grid of points generated using the previously demonstrated mechanism. The initiation point for the stimulus propagation is randomly selected within the grid.
Every point is then generated by simulating the time-evolving solution of the Fitzhugh-Nagumo model, with a final time of $T = 500\ ms$ and a time step size of $dt = 0.5$. We utilize the theta method for the solution, implemented via the ThetaMethod class. The solver parameters are outlined in the file `parameters_FN.txt`.

7.4. Neural Networks

The following step is to train the neural networks associated with the LED class: the autoencoder and the Recurrent Neural Network (RNN). The autoencoder is necessary when the dimensionality of the dataset increases because, if in the Van der Pol case the number of variables to be predicted is only 2, in this case we need to forecast the behaviour of each cell in a 20x20 grid, and the training of a recurrent network on a dataset with that many features is not ideal, since the accuracy is lower and the training time is much higher. For this reason we opted to encode each snapshot of a sample into a smaller dimension, and to do that we use an autoencoder.

7.4.1. Autoencoder

The autoencoder is composed of two sub-models: an encoder and a decoder. When given an input, in our case a 20x20 matrix, it is trained to be able to reconstruct the input itself, but between the encoder and the decoder there is a layer of neurons, the size of which we called `latent_dim` which acts as a sort of bottleneck, forcing the decoder to be able to reconstruct the input from a much smaller set of variables. We use this property of the autoencoder to reduce the dimension of the data, since calling the encoder model on each snapshot produces an encoded dataset, which has dimension `latent_dim` instead of 20x20.
Figure 15 shows the general structure of an autoencoder.

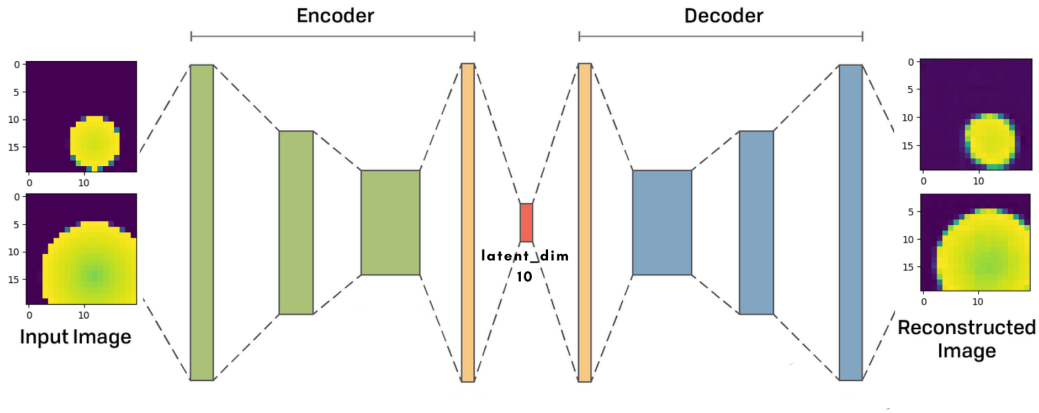


Figure 15: Autoencoder.

The model, as shown in Figure 16 is composed of convolutional and pooling layers, followed by dense layers. The user can decide the number and size of each layer, and the model which gave the best results is shown in figure 16.

Model: "Encoder"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 20, 20, 8)	152
max_pooling2d (MaxPooling2D)	(None, 10, 10, 8)	0
conv2d_1 (Conv2D)	(None, 10, 10, 16)	1168
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 16)	0
conv2d_2 (Conv2D)	(None, 5, 5, 32)	4640
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 32)	0
flatten (Flatten)	(None, 128)	0
dense (Dense)	(None, 64)	8256
dense_2 (Dense)	(None, 64)	4160
dropout (Dropout)	(None, 64)	0
dense_4 (Dense)	(None, 128)	8320
dropout_2 (Dropout)	(None, 128)	0
dense_6 (Dense)	(None, 10)	1290

Total params: 27986 (109.32 KB)
Trainable params: 27986 (109.32 KB)
Non-trainable params: 0 (0.00 Byte)

Model: "Decoder"

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 128)	1408
dropout_3 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 64)	8256
dropout_1 (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 64)	4160
reshape (Reshape)	(None, 2, 2, 16)	0
conv2d_transpose_3 (Conv2DTranspose)	(None, 5, 5, 32)	4640
conv2d_transpose_2 (Conv2DTranspose)	(None, 10, 10, 16)	4624
conv2d_transpose_1 (Conv2DTranspose)	(None, 20, 20, 8)	1160
conv2d_transpose (Conv2DTranspose)	(None, 20, 20, 2)	66

Total params: 24314 (94.98 KB)
Trainable params: 24314 (94.98 KB)
Non-trainable params: 0 (0.00 Byte)

Figure 16: Summary of encoder and decoder for the Fitzhugh-Nagumo model.

The training is done using the mean squared error as loss function, and the Adam optimizer from the Keras library. In order to prevent overfitting we add `batch_normalization` layers between both convolutional and dense layers, which increase the accuracy.
The final model has `latent_dim` $= 10$, which is a good compromise between the need of the autoencoder of a high latent dimension in order to better reconstruct the input, and the need of the RNN of a low input dimension in order to be able to forecast.
Figures 17 and 18 show the performance of the autoencoder for fixed time instants and for a fixed particle, respectively.

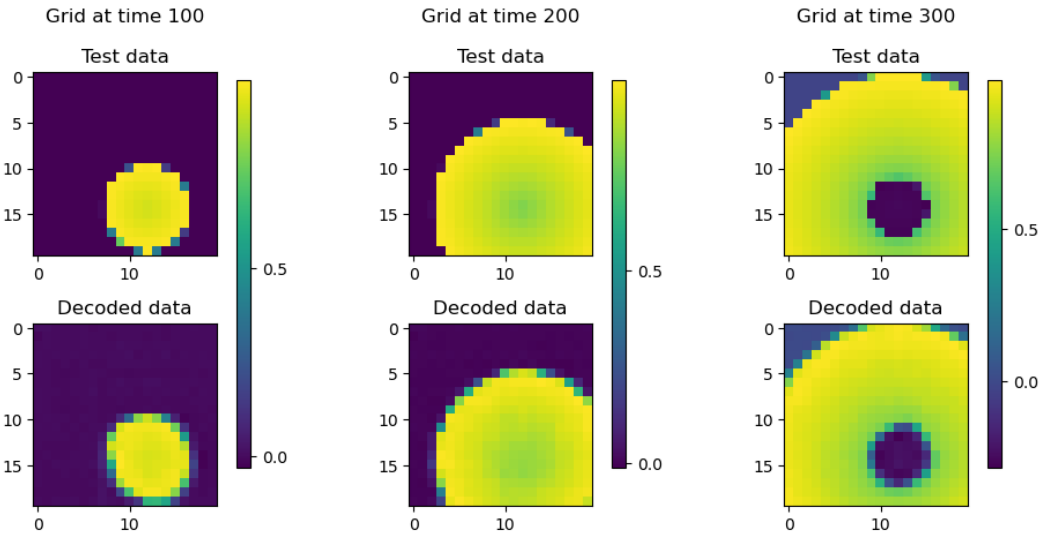


Figure 17: Autoencoder reconstruction of the grid at different fixed time instants.

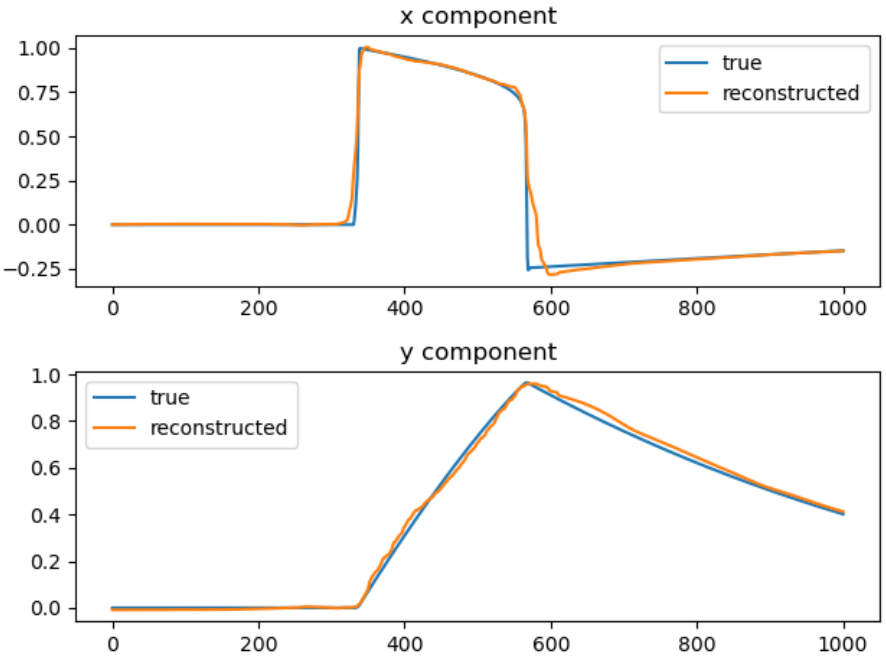


Figure 18: Autoencoder reconstruction of the trend over time of a fixed particle ($x_0 = 2, y_0 = 1$).

To reproduce the results of the autoencoder for the Fitzhugh-Nagumo model, refer to the Jupyter notebook `test_autoencoder` from the folder `examples/Fitzhugh-Nagumo`.

7.4.2. Recurrent Neural Network

The training is done using the mean absolute error loss function and the Adam optimizer from the Keras library. Also in this case we encountered some overfitting problems during the first attempts in training the model, so we decided to add the possibility of adding some `dropout` and `batch_normalization` layers between LSTM layers.
After conducting various experiments and exploring different configurations, the most optimal arrangement is found to be the following:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 bidirectional (Bidirection  (None, 200, 64)           11008
 al)

 batch_normalization (Batch  (None, 200, 64)           256
 Normalization)

 bidirectional_1 (Bidirecti  (None, 128)               66048
 onal)

 dense (Dense)               (None, 64)                8256

 batch_normalization_1 (Bat  (None, 64)                256
 chNormalization)

 dense_1 (Dense)             (None, 32)                2080

 batch_normalization_2 (Bat  (None, 32)                128
 chNormalization)

 dense_2 (Dense)             (None, 10)                330

=================================================================
Total params: 88362 (345.16 KB)
Trainable params: 88042 (343.91 KB)
Non-trainable params: 320 (1.25 KB)
_________________________________________________________________
```

Figure 19: Summary of the RNN for the Fitzhugh-Nagumo model.

An interesting fact we noticed while training the RNN is that the dependence between the dimension of the input and the performance of the network is not as trivial as "smaller is better", since when the dimension is too low the model, while training faster, has a worse forecasting capability than a model trained on a higher dimension input. Also for this reason we decided to stick with `latent_dim` $= 10$, since we found that this is the best compromise between training speed and overall accuracy.
Figure 20 shows the performance of the RNN on the test set for two variables in the latent space.

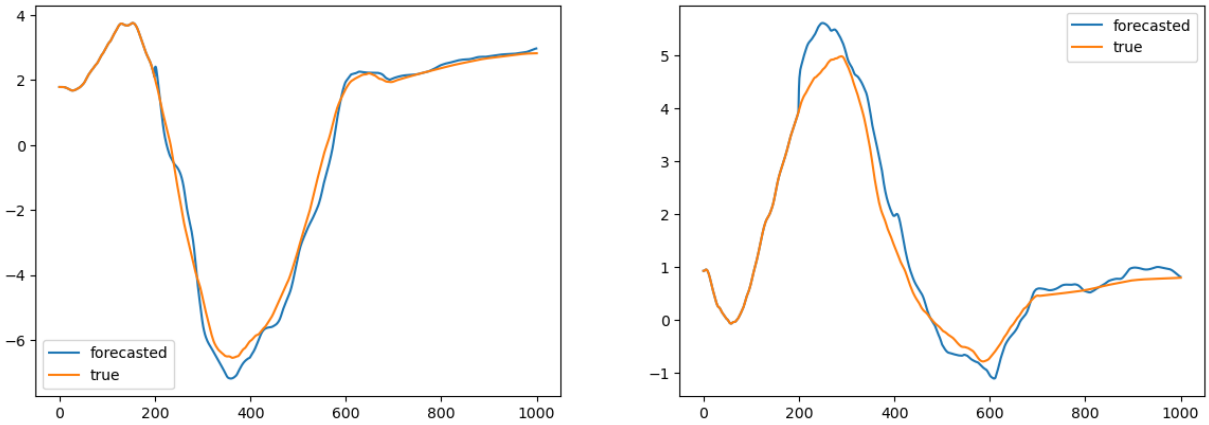
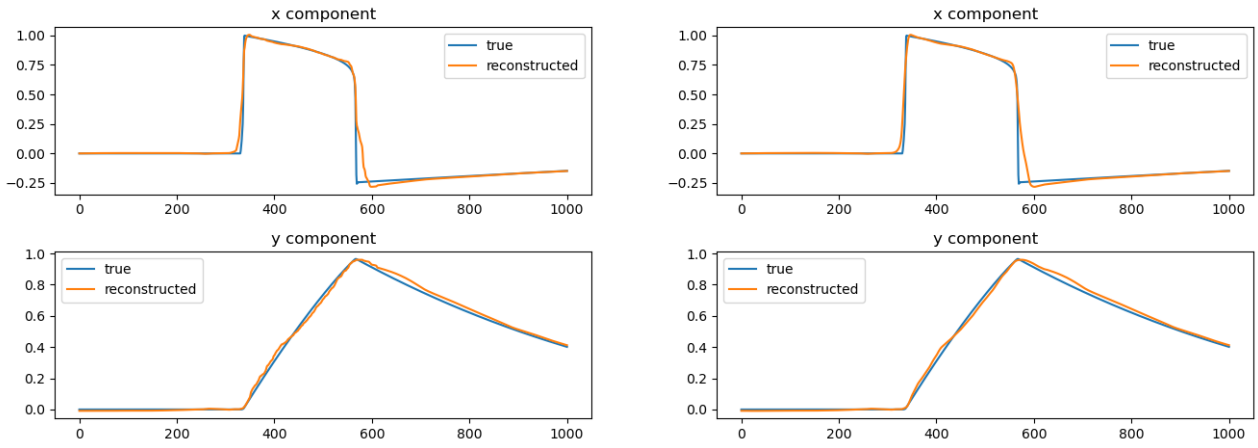


Figure 20: RNN prediction on the test set.

7.5. Smoothing

After training both the autoencoder and the RNN, when putting them together in the LED class, we discovered that one of the factors that limits the performance the most is the lack of continuity between different time instants. This is due to the fact that the autoencoder is trained on single snapshots of the solution, and therefore the encoding process is not a continuous function, which translates in a staggered encoded dataset and gives some problems to the RNN, especially when forecasting on new data.
In order to solve this problem we added the possibility of smoothing the data immediately after encoding, and training the RNN on this smoothed data. This is possible due to the fact that the decoder, when decoding the smoothed encoded data, gives a close enough reconstruction of the original input, as seen in figure 21.



(a) Raw. (b) Smoothed.

Figure 21: Autoencoder reconstruction with and without the smoothing.

At first we tried to smooth the data using a simple moving average, but we saw that it is not feasible for our problem, since the data has a spike that must be detected from the RNN, and a moving average reduces the slope which happens when the signal reaches a particular cell. For this reason we applied a particular version of a low-pass filter called Savitzky-Golay filter [13] which approximates the function as a polynomial through a least-squares estimation of the coefficients. After applying the smoothing filter to the encoded data we trained the RNN again and we noticed a better accuracy which is translated to a better accuracy for the LED class.

7.6. LED

After the training and the evaluation of the two networks separately, we ensemble them through the LED class.

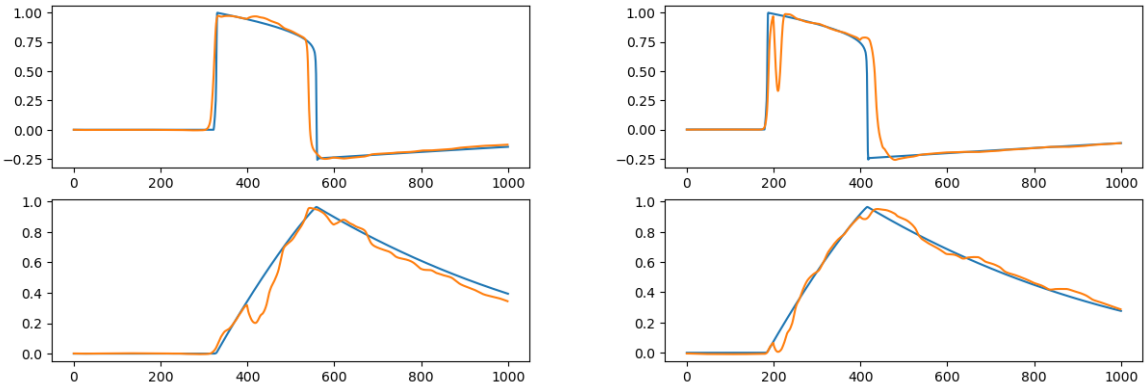


Figure 22: Reconstruction of the signal through the LED in two different points of the grid.

Figure 22 shows that there are still some problems in the LED results.
To enhance its performance, several approaches can be considered.
Firstly, optimizing the hyperparameters of both networks could lead to improvement. Another strategy involves jointly training the RNN and the decoder part to mitigate the accumulation of errors between them. Lastly, in [11], the authors propose a novel method for evolving the dynamical system with the LED. This approach incorporates error computation, creating a hybrid system between surrogate model and numerical solver.

To reproduce the results, run the `main` file inside `examples/Fitzhug-Nagumo`.

8. Asymmetric Fitzhugh-Nagumo

Starting from a given signal within a grid, the user could want to evaluate the propagation of the signal in different points of the grid with respect to the original one.
For this purpose, we introduce the asymmetric autoencoder, as explained in 4.2.

With this tool, starting from a 10x10 grid on the unit square, the user is able to predict the signal on a finer grid, specifically a 20x20 grid.
The training is done using the mean absolute error loss function and the Adam optimizer. The final composition of the asymmetric autoencoder is the following:

Model: "Encoder"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 10, 10, 8)	152
max_pooling2d (MaxPooling2D)	(None, 5, 5, 8)	0
conv2d_1 (Conv2D)	(None, 5, 5, 16)	1168
max_pooling2d_1 (MaxPooling2D)	(None, 2, 2, 16)	0
conv2d_2 (Conv2D)	(None, 2, 2, 32)	4640
max_pooling2d_2 (MaxPooling2D)	(None, 1, 1, 32)	0
flatten (Flatten)	(None, 32)	0
dense (Dense)	(None, 32)	1056
dense_2 (Dense)	(None, 32)	1056
dropout (Dropout)	(None, 32)	0
dense_4 (Dense)	(None, 64)	2112
dropout_2 (Dropout)	(None, 64)	0
dense_6 (Dense)	(None, 10)	650

Total params: 10834 (42.32 KB)
Trainable params: 10834 (42.32 KB)
Non-trainable params: 0 (0.00 Byte)

Model: "Decoder"

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 64)	704
dropout_3 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 32)	2080
dropout_1 (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 32)	1056
reshape (Reshape)	(None, 1, 1, 32)	0
conv2d_transpose_4 (Conv2DTranspose)	(None, 2, 2, 32)	9248
conv2d_transpose_3 (Conv2DTranspose)	(None, 5, 5, 16)	4624
conv2d_transpose_2 (Conv2DTranspose)	(None, 10, 10, 8)	1160
conv2d_transpose_1 (Conv2DTranspose)	(None, 20, 20, 2)	18
conv2d_transpose (Conv2DTranspose)	(None, 20, 20, 2)	18

Total params: 18908 (73.86 KB)
Trainable params: 18908 (73.86 KB)
Non-trainable params: 0 (0.00 Byte)

Figure 23: Summary of encoder and decoder for the asymmetric Fitzhugh-Nagumo.

Figures 24 and 25 show the performance of the asymmetric autoencoder.

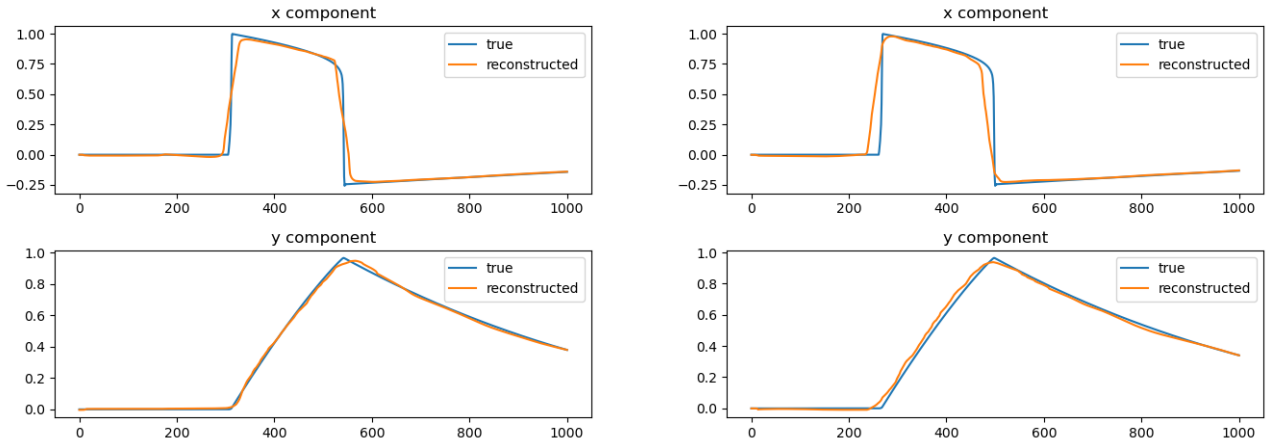


Figure 24: Asymmetric autoencoder reconstruction of the trend over time of two fixed particle, ($x_0 = 2, y_0 = 3$) and ($x_0 = 4, y_0 = 4$).

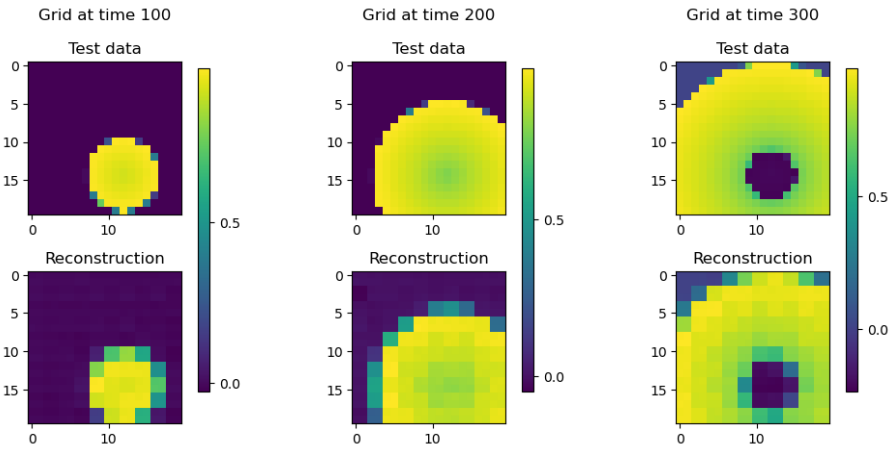


Figure 25: Asymmetric autoencoder reconstruction of the grid at different fixed time instants.

A customized RNN is trained for this case-study. Ensembling the autoencoder and the RNN, we instantiate the LED object and run the prediction. Figure 26 shows the results.

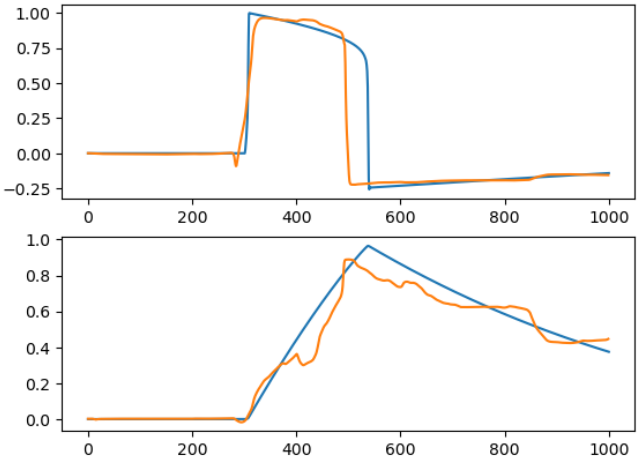


Figure 26: Reconstruction of the signal through the LED employing the asymmetric autoencoder.

The same considerations for improving performance, as in the case of the classical autoencoder, apply here. In fact, while the general trend of the signal is catch, is evident that a more accurate prediction is needed.

To reproduce the results for the evaluation of the autoencoder and the LED run the Jupyter notebooks inside the folder `examples/Fitzhugh-Nagumo asymmetric`.

9. Conclusions

We have presented the code structure to support the construction and training of the LED framework outlined in [16].
The final code also facilitates the creation of the datasets based on the solution of ordinary differential equations. The entire structure enables new users to build their own case-studies, starting from dataset generation, progressing to the training of the autoencoder and recurrent neural network, and concluding with the reproduction and evaluation of the LED.
Thanks to the Abstract Class Python tool and the inheritance mechanism, we have built a code structure that is a balanced compromise between easy-usage and customization.
The implementation of two examples, Van Der Pol and Fitzhugh-Nagumo, demonstrates the effective user-friendliness of the code. The results obtained for these dynamical systems show the LED framework's capability to capture system dynamics while achieving increased computational speed.

Regarding the structure supporting data generation, further enhancements could involve adding additional solvers. Given the current inheritance structure of the GenerateParticle class, incorporating such extensions is straightforward.

To improve the accuracy of the prediction of the LED, several approaches can be considered. Additional enhancements involve training the autoencoder and the RNN jointly to mitigate errors during the transition between the two. A better performance for the autoencoder and the RNN could be achieved through a further investigation of the hyperparameters.
Further developments include also the implementation of the Adaptive LED. It was introduced in [11] to overcome the lack on continuous training, which limits the ability of the network to change dynamics, to give the possibility to restart a computer simulation from any time point, to quantify prediction uncertainty and other issues. This new system is based on the alternation of the computational solver and the surrogate model composed of the autoencoder and the RNN.

References

[1] ABC class documentation. https://docs.python.org/3/library/abc.html.

[2] Keras documentation. https://keras.io/api/.

[3] Multiprocessing documentation. https://docs.python.org/3/library/multiprocessing.html.

[4] NumPy documentation. https://numpy.org/doc/1.26/.

[5] Project LED, GitHub repository. https://github.com/ibismanu/PACSproject_LED.

[6] Rnn training mechanism. https://ml-lectures.org/docs/unsupervised_learning/ml_unsupervised-2.html.

[7] Tensorflow documentation. https://www.tensorflow.org/api_docs.

[8] B. Van der Pol. On "relaxation-oscillations". *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 2 (11)*, 1926.

[9] Arieh Iserles. Mathematical tripos part ii, lent 2005. Numerical Analysis, Lecture 9.

[10] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge University Press, 1996.

[11] I. kicic, P.R Vlachas, G. Arampatzis, M. Chatzimanolakis, L. Guibas, and P. Koumoutsakos. Adaptive learning of effective dynamics: Adaptive real-time, online modeling for complex systems. 2023.

[12] Saul A.; Vetterling William T.; Flannery Brian P. Press, William H.; Teukolsky. *Section 17.1 Runge-Kutta Method, Numerical Recipes: The Art of Scientific Computing (3rd ed.)*. Cambridge University Press, 2007.

[13] M.J.E. Savitzky, A.; Golay. Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry*, 1964.

[14] Sepp Hochreiter; Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

[15] William E. Sherwood. Fitzhugh-nagumo model. *Encyclopedia of Computational Neuroscience*, 2014.

[16] P.R Vlachas, G. Arampatzis, C. Uhler, and P. Koumoutsakos. Learning the effective dynamics of complex multiscale systems. *Nature Machine Intelligence 4*, 2020.