UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

TREBALL DE FINAL DE GRAU

Grau en Matemàtiques

Machine Learning techniques for optimal worm-like motion

Autor: Albert Jiménez Blanco

Director: José Javier Muñoz Romero

Departament: Matemàtiques

Convocatòria: Maig 2023

I admit that mathematical science is a good thing. But excessive devotion to it is a bad thing.

Aldous Huxley

Acknowledgements

What makes possible any project cannot be summarized in a paragraph, as everything heard, observed, learnt or experienced has an influence on an individidual. So, if being faithful, I started thanking all the small contributions, my hand would not raise from the computer in a long time. For simplicity's sake, then, I will only mention those people who had a deeper and more durable effect. First, and without any doubt, I wanted to thank my family, specially my parents, for making me the person I am, for teaching me to love knowledge and to think beyond myself. I need to mention as well my friends both from here and from Louvain-la-Neuve, in particular Dani, who told me to do this thesis. I would also like to thank everyone that has shown me the beauty of mathematics: my high school teachers Albert and Marc Oriol, the university professor that convinced of studying mathematics.... And finally, I would like to give a special shout out to the director of this thesis, José Javier Muñoz, not only for proposing such an interesting topic, but also for guiding me through all the process.

And, as I mentioned before, with this list I have left a lot of people out, so I'll just add some names that also deserved to be mentioned, being conscious that, again, I will be falling short: Salva, Lidia, Julia, Gabriel, Elian, Kora, Àlex, Giorgia, Agustina, Borja, Denis, Andrea, Gerard, Oriol, Anna, Sara, Xavi, Martí, Pau, Míriam, Juan, Mercè...

Albert Jiménez Blanco

Abstract

In a context of popularization of Artificial Intelligence techniques, this project apllies its most known implementation, neural networks, to try to understand the dynamics underlying the motion of the nematode *C.elegans*. Dense, convolutional and recurrent networks will be tried to predict moments in base of positions and validate the current theory of these worm's motion. The capacity of Neural Networks to predict muscle activity, i.e. bending moments, from both synthetic and experimental positions will prove to be very good. However, more investigation would be needed in order to reach a definitive answer.

Resum

En un context de popularització de les tècniques d'Intel·ligència Artificial, aquest treball es proposa aplicar la versió més coneguda d'aquestes, les xarxes neuronals, per tractar comprendre la dinàmica del tipus de cuc *C.elegans*. Xarxes denses, convolucionals i recurrents van ser provades per tractar de predir moments de flexió a través de posicions i validar l'actual teoria sobre el moviment d'aquests cucs. La capacitat de les Xarxes Neuronals de predir moments a través de posicions sintètiques i experimentals demostrarà ser molt bona, tot i que caldria més investigació per aconseguir una resposta definitiva.

Resumen

En un contexto de popularización de las técnicas de Inteligencia Artificial, este trabajo se propone aplicar la versión más conocida de estas, las redes neuronales, para tratar de comprender la dinámica del tipo de gusano *C.elegans*. Redes densas, convolucionales y recurrentes serán probadas para tratar de predecir momentos de flexión a través de posiciones y validar la actual teoría sobre el movimiento de estos gusanos. La capacidad de las Redes Neuronales de predecir momentos a través de posiciones sintéticas demostrará ser muy buena, aunque haría falta más investigación para llegar a una respuesta definitiva.

Contents

1 Introduction

In the last few years, the field of Machine Learning has delivered some very interesting results and, because of that, it has gained a lot of attention. Even though Machine Learning is a broad domain, lately, a specific kind of algorithm stands out amongst the others: Neural Networks. However, these algorithms alone cannot explain the high levels of performance observed in many of the programs based on their structure. To function properly, they need massive quantities of data and/or a great computational power. And, thanks to the Internet and an improvement of the hardware used by computers, this two conditions have been met[Mit20].

Under these circumstances many people have been trying to apply Neural Networks in different areas with varied success. Amongst the success cases, one can find, for example, image classification algorithms, face-recognition programs and several applications related to Natural Language Processing, as the lately popularized chatbot ChatGPT. However, these algorithms are not magical tools that can solve all of our problems and, clearly, there don't yet have a human-like understanding of the data that they process. As it has been discussed by Melanie Mitchell in "Artificial Intelligence: A Guide for Thinking Humans", Machine Learning and, in particular, Neural Networks, have proven to be very successful in narrowly designed tasks and, preferably, under controlled environments. Luckily, the problem that will be tackled in this Final Degree Project adjusts to these parameters.

1.1 Objectives

This thesis is part of a larger effort to understand the motion dynamics underlying the movement of worm-like organisms, ideally finding equations describing these dynamics sufficiently well. The current project has the goal of using Machine Learning techniques in order to gain insights on the topic and check whether the current hypotheses hold, or some new ones should be considered. In particular, the objective will be to train a Neural Network that, based on positions, outputs estimated moments.

Even though the main motivation to carry out this project is scientific and curiosity is the motor guiding it, one can easily imagine several applications of the understanding of the motion of worm-like organisms. A clear example of those could be medical worm-like robots, which could help in many situations as, for instance, in carrying proteins through the bloodstream.

1.2 The Problem

The central topic of this project, as it is outlined by its title, is worm-like motion. As it has been mentioned in the previous section, the goal is to validate or refute the current hypothesis regarding worm's motion. It must be clarified, though, that the subjects of study are not all kinds of worm-like beings, but only worm-like microorganisms. More particularly, the study will focus on a preferred amongst scientists: *C. elegans*[1]. That is, the empirical data that will be used to check the model's accuracy will be extracted from footage of moving worms of this kind.

Along this thesis, the code presented by David Doste in his Master's Thesis[Poy20] will be used. There, a mathematical model for the motion of the worm was developed and, then, implemented in Matlab. The model is based on the hypothesis that the frictional law that allows movement is wet friction, as it depends on velocity and not on displacement, as in a fluid.

[1] *C.elegans* are characterized for being very simple but, at the same time, complex enough to have a nervous system and for being transparent.

2 Background on Machine Learning

In order to properly understand the techniques that will be used in the thesis and to rigorously lay the foundation of its results, the main concepts behind Machine Learning and, particularly, Neural Networks will be explained.

A first thing to take into account is that Neural Networks are a part of Machine Learning and, Machine Learning is, in turn, a subset of Artificial Intelligence. In consequence, they navigate in the long-lasting human dream of creating an intelligent machine. In this project, the slippery word of "intelligence" translates to capturing the underlying dynamics of worm-like organisms by deducing, from the footage of moving worms, the moments that the worm has exerted to move as it has.

The specific traits of Machine Learning can be found on the second word of its name: Learning. The idea is that the algorithms of this kind "learn" from experience, which basically means updating certain parameters, to perform a given task. It is important, though, to keep in mind that this "learning" is, for now and by a large distance, very different from the one experienced by humans[Mit20].

2.1 Supervised and Unsupervised Learning

There are two ways in which a machine can "learn": supervised and unsupervised learning. In the first case, a certain amount of labeled data is needed[Meh21]: The algorithm will update its parameters in such a way that, given a certain input, it returns the best approximation that it can to its output. In the second, instead of basing its learning on fitting as best as possible to the given data, the algorithm will perform actions in an environment and, when it gets to the desired outcome, it will receive a reward, which will allow it to "learn". For the purpose of this project, the first kind of learning will be used.

There exist many Machine Learning algorithms that base its "learning" on fitting their predictions to the correct outputs. Some of the most prominent are Linear regressions, Support Vector Machines, Random Forests and, most notably, Neural Networks. What characterizes each one of them is the way in which data is used to update the corresponding parameters. In the next section, the chosen approach for this thesis (Neural Networks) will be explained.

2.2 Neural Networks

Inspired by the way the brain works, with loads of connected neurons which share information in the form of electric impulses, their structure is composed of several layers of units. From a general overview, all Neural Networks have 3 clearly defined parts: an input layer, a central part with some hidden layers and an output layer. The main idea is that the algorithm will receive a certain input, which will be processed by the successive hidden layers and, finally, it will return a certain output. Then, the two fundamental elements that compose the structure of Neural Networks are layers and units.

2.2.1 Layers and units

The best way to understand what a **layer** is, is through an image of the structure of a Neural Network. As it can be seen in Figure 1, a layer is just a set of units (the orange circles) that work together and are in a specific depth of the network. That is, in this particular Neural Network there are 3 layers with, respectively, 3, 4 and 2 units.

One can think of a **unit** as having a similar role to the one a neuron would have in a brain. That is, units receive inputs and, after processing them, return a certain output, analogously to how neurons receive electric inputs and then, after processing the signal, send -or not- a signal to other neurons.

As it can be seen in the figure, each unit can be connected to many others, both receiving or sending a signal. More precisely, this signal will be a numerical value. Generally, Neural Networks propagate forward, that is, the signal will go from one layer to the next one. In any case, the

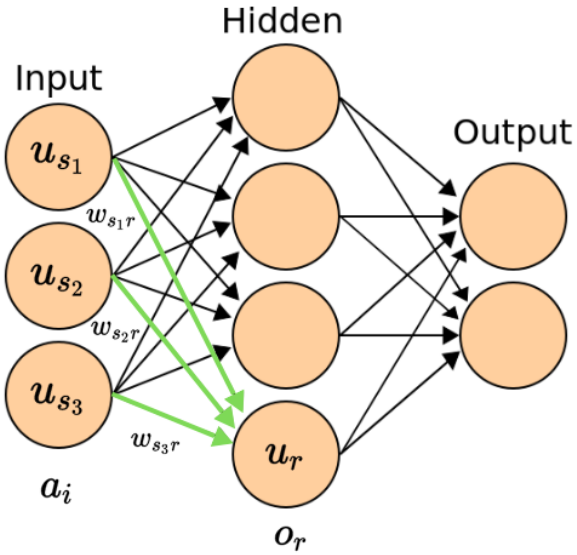


Figure 1: Simple Neural Network. Image extracted from Wikipedia with some changes

process will always start in the input layer, where a set of n^2 values will be fed to the Network. To understand their functioning better, it is useful to dissect Neural Networks into two stages: the training one and the execution one.

2.2.2 Weights and biases

As it has already been mentioned, units send and receive numerical values. Nevertheless, it still remains a mystery how this value is processed inside of the unit. To be able to understand this, it is necessary to introduce two new objects: **weights** and **biases**. Weights will be associated to two connected units, a sender and a receiver, and biases only to the receiver unit.

Let's take a receiving unit u_r which is connected to l senders u_{s_1}, u_{s_2}, ... u_{s_l}. Each unit will be sending u_r a numerical value a_i. Let's call w_{s_ir} the weight associated to the ordered pair (u_{s_i}, u_r) and b_r the bias corresponding to u_r. Then, the value o_r that u_r will output is

$$o_r = \sum_{i=1}^{l} a_i w_{s_ir} + b_r \tag{1}$$

That is, it will be a linear combination of the inputs, where the weights act as the coefficients multiplying the input values and the bias is the independent term.

When training a Neural Network, these weights and biases are the parameters that are going to be updated to improve the performance of the algorithm[Meh21]. One could say that, so far, this procedure looks quite similar to a Linear Regression, and, in fact, it would be a relevant observation. If nothing else was added to this model, the output values would end up being linear combinations of the inputs and, as the goal would be to have the most accurate predictions (reduce the error between prediction and measured output), if the metric used for improvement was the Minimum Squared Error, the resulting model would be, precisely, a Linear Regression. Before searching for the last element that is missing in the model, let's check that, indeed, without any other elements, the outputs of Neural Networks would just be linear combinations of their inputs. We will prove this result by induction on the number of hidden layers. Let's call the input values a_i, the output

[2] n denotes the number of units in the first layer.

values o_j and suppose that the Neural Network has 0 hidden layers. Then,

$$o_j = \sum_{i=1}^{l} a_i w_{ij} + b_j \tag{2}$$

In consequence, the output values are linear combinations of the inputs. Let's prove that if the result holds for n hidden layers it will also hold for $n+1$ hidden layers. To do so, let's break the Neural Network of $n+1$ hidden layers into two parts: in one side the input layer and, in the other, the rest of layers. In the second part, we can consider the first hidden layer to be an input layer (because it is now the first layer of the new Network created by extracting the original input layer). By hypothesis, the outputs are lineal combinations of the inputs. That is,

$$o_j = \sum_{i=1}^{m} \alpha_i c_i + b_j \tag{3}$$

where c_i indicates the values in the first hidden layer (the one which became the new input layer). However, in the original Neural Network, every c_i is just a linear combination of the input values. Then,

$$o_j = \sum_{i=1}^{m}(\sum_{k=1}^{l} a_k w_{ki} + b_i)\alpha_i + b_j = \sum_{k=1}^{l}(\sum_{i=1}^{m} w_{ki}\alpha_i)a_k + (\sum_{i=1}^{m} b_i\alpha_i + b_j) \tag{4}$$

As $\sum_{i=1}^{m} w_{ki}\alpha_i$ and $\sum_{i=1}^{m} b_i\alpha_i + b_j$ are combinations of coefficients, they are also coefficients. Thus, for $n+1$ hidden units the result also holds and our hypothesis is proven.

2.2.3 Activation functions

The elements that were missing in the previous design were activation functions, which introduce non-linearities. When the inputs have been processed and a numerical value is going to be sent, it passes first through an activation function, which changes its value in, usually, a non-linear way. Therefore, the output o_r ends up being

$$o_r = A(\sum_{i=1}^{l} a_i w_{s_i r} + b_r) \tag{5}$$

where A indicates the chosen activation function. Some examples of common activation functions are:

$$S(x) = \frac{1}{1+e^{-x}} \qquad R(x) = \max(x, 0) \qquad T(x) = \frac{1-e^{-2x}}{1+e^{-2x}} \tag{6}$$

Sigmoid **ReLu** **TanH**

2.2.4 Training stage

Now that the basic architecture of Neural Networks is clear and all of its main elements have been described, let's analyse how the learning process is carried out. First, some initial values are assigned to all the weights and biases of the network. This can be done randomly but, sometimes, one might have previous knowledge allowing to tune the initial values more finely. Then, the same updating process will be repeated over and over again until meeting a certain stopping criteria.

An input, with a known output, is passed through the network, which makes a prediction. Then, the prediction is compared with the real output and the error propagates backwards, by distributing the "guilt" to the different weights and biases. Their values will be updated according to their respective influence in the final error. It is important to keep in mind that the "guilt" will be computed in such a way that the updates it causes will minimize the error. That is, with the

new parameters and the same input, the new prediction aims to have a smaller error respect the targeted value.

The first thing to decide is how the error between predicted and real outputs will be computed. That is, to choose which **loss function** will the algorithm use to quantify the error of the Neural Network's prediction respect its targeted (real) output. Most implementations use the Mean Squared Error, which always returns a positive value.

As said before, the guilt will then be distributed via **backpropagation**. In practice, this means that the partial derivatives of the error function respect each parameter will be computed in a recursive way, starting by the output layer and using the chain rule. The next step will be to update the parameters using this information. That is, as the goal is to minimize the error, the parameters should be updated proportionally to the partial derivatives. Imagine the partial derivative respect a weight w to be positive. This would mean that increasing its value would also increase the error. Then, what should be done is decreasing its value, and doing so proportionally to the partial derivative.

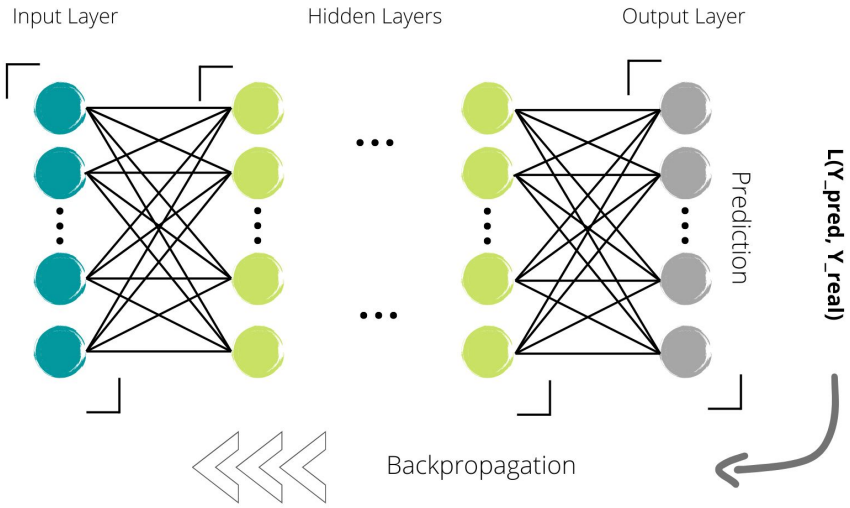


Figure 2: Loss function and Backpropagation

Now, one of the most common implementations of the backpropagation algorithm, using the **Stochastic Gradient Descent** method, will be discussed. The theory explained here can be found at [Meh21]. Let $l = 1, 2...L$ be the variable indexing the different layers. As discussed before, the outputs at layer l are computed as

$$O_j^{(l)} = A(Z_j^{(l)}) \quad \text{with} \quad Z_j^{(l)} = \sum_k w_{jk}^{(l)} O_k^{(l-1)} + b_j^{(l)} \tag{7}$$

Then, computing the corresponding partial derivatives and applying the chain rule, one gets to the following update formulae

$$\delta w_{mn}^{(l)} = \eta \delta_m^{(l)} O_n^{(l)} \quad \text{and} \quad \delta b_m^{(l)} = \eta \delta_m^{(l)} \tag{8}$$

where

$$\delta_j^{(l-1)} = \sum_i (t_i - O_i^{(L)}) \frac{\partial O_i^{(L)}}{\partial O_j^{(l-1)}} A'(Z_j^{(l-1)}) \tag{9}$$

indicates the error at layer $l-1$, t_i is the value of component i of the vector of targeted values $\boldsymbol{t}$, the lost function used was $H = \frac{1}{2}\sum_i (t_i - O_i)^2$ and $\eta > 0$ indicates the chosen **learning rate**. The

weights and biases would then be updated by adding them the values computed using equation 8. That is,

$$w_{mn}^{(l)'} = w_{mn}^{(l)} + \delta w_{mn}^{(l)} \quad \text{and} \quad b_{m}^{(l)'} = b_{m}^{(l)} + \delta b_{m}^{(l)} \tag{10}$$

The method previously described allows to efficiently compute the partial derivatives in order to modify the parameters of the network in the direction of maximum descent. The Stochastic Gradient Descent is an **optimizer**: a way to optimize the parameters of the Neural Network to achieve better results. Nowadays, the methods that are mostly used are improved versions of this one, which use, for instance, varying learning rates (the parameter that quantifies the amount of change applied in each iteration). In particular, the Adam optimizer is the most popular in the python environment and is the one that will be used in the thesis.

2.2.5 Execution stage

The training will go on until a certain metric is met. At this point, the parameters will stop updating and, in theory, this fixed values will be codifying the necessary "knowledge" on the topic to perform their regression task sufficiently well. Generally, given a new input of data, the algorithm will send it forward, processing it at each layer until arriving to the last one: the output layer. After that, the algorithm will return these last values, which will be its prediction. As it has been described before, for processing the data, the only thing that the Network does is the following: it multiplies the input coming from the previous layer by its weights matrix $\boldsymbol{W^k}$, it adds the bias vector $\boldsymbol{B}^k$ and finally applies the activation function $\boldsymbol{A}^k$. Then, this activation value will be sent to the next layer and the process will be repeated. Depending on the layer where the processing is happening, two cases can be distinguished:

$$1) \quad \boldsymbol{A^k(W^k A^{k-1} + B^k)} \quad \text{if k} \geq 2 \qquad\qquad 2) \quad \boldsymbol{A^k(W^k I + B^k)} \quad \text{if k} = 2 \tag{11}$$

That is, in the second the layer the input is directly the Network's input and, in the rest, it is the output of the previous layer.

2.2.6 Types of layers

Another important aspect to take into account when designing a Neural Network is the structure that it will have, which basically refers to the layers that it will be built upon and their distribution in the Network. The original and simplest kind of layers that exist are the "Dense" or "Fully connected" layers, which, as their name indicates, connect all the input units on the previous layer with all the units in the current one. This kind of layer can be seen, for instance, in Figure 1, which is composed of two fully connected layers.

With time, however, other kinds of layers and structures were proposed, which allowed to model more complex topics by adding new ways of processing the data. Due to the nature of the data for this project, **Convolutional** and **Recurrent** layers seem to be two good options to take into account.

Convolutional layers

The first ones, which are behind the success of the Image-related machine learning algorithms[ON15], is based on the idea of analysing the input data, which in the case of images corresponds to matrices of numerical values, by parts and using a common pattern. Their name comes from the fact that the same convolution kernel or filter is applied to all the elements in the input. Sticking to the case of an image, that would mean that the same convolution would be applied to all the pixels of the image, causing an invariance that allows to capture features irrespective from their position in the input data, as the filters used are always the same.

In mathematical terms, a Convolution is an operation specific to two functions, producing a third one expressing how the shape of one is modified by the other. Let f and g be two functions and $f * g$ its convolution. Then,

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau - t)g(\tau)d\tau \tag{12}$$

One could interpret here the second function g as a weight function that, depending on the value of t, would be giving more importance to certain parts of the function f or to others.

However, the interesting case for Machine Learning is the discrete version of this and, more specifically, that where both functions f and g have a finite domain. Which this implies is that, instead of talking about functions, one can also think of convolutions as being applied vectors. Calling $\boldsymbol{v}$ and $\boldsymbol{w}$ to these new vectors, the previous expression would translate then to

$$(\boldsymbol{v} * \boldsymbol{w})(k) = \sum_{i,j|i+j=k} v(i)w(j) \;\; \forall k \tag{13}$$

where k can be all the possible sums of indices. The result of this convolution would be a new vector of length $n + m$ - 1, being n the length of $\boldsymbol{v}$ and m the length of $\boldsymbol{w}$. When implementing this on a Neural Network, the same filter, which in this case would correspond to $\boldsymbol{w}$, would be applied to several parts of the input.

Recurrent layers

This second kind of layer is widely used for problems where the input data shows some temporal behaviour[Sch19], as the idea is that the data will be processed in various steps and, for each processing, the outputs from previous steps will be used as inputs for the next one. A typical representation of a Recurrent layer draws a self-pointing arrow, indicating the recursion in the process

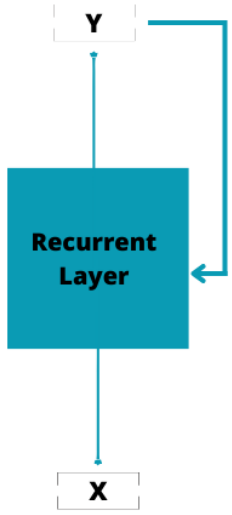


Figure 3: Graphical representation of a recurrent layer

As it is shown by the image, the input that the layer will receive at step i are X_i and Y_{i-1}. This process is repeated n times, where n corresponds to the numbers of steps in which the input data is divided. What is very important to take into account is the fact that the weights are the same for all the time steps: the only thing that will change is the input that the neutwork receives.

3 Mathematical Model

3.1 The model

Even though the mathematical model used in this thesis was developed by David Doste Poy on his master's thesis[Poy20] and is explained there in detail, it will be briefly discussed in this section. The first necessary step in order to model the worm's dynamics is discretization: the worm will be divided into n segments (corresponding to $n+1$ nodes).

Due to its elastic behaviour, two elastic forces will be considered: stretching and bending forces, both coming from elastic potential energies. As their names indicate, the first elastic potential will depend on how much is the worm bending and, the second, on how much it stretches. What this means for the equations is that the interesting parts will be the curvatures between consecutive segments and their elongation. The specific implementation of these ideas is the following: we define bending and stretching elastic potencials W^b and W^s respectively, expressed as,

$$W^b = \frac{1}{2} k_b \sum_{i=2}^{n} \theta_i^2 \tag{3.1}$$

where k_b is the elastic bending constant and θ_i is the angle that form the line that goes through $\boldsymbol{x_{i-1}}$ and $\boldsymbol{x_i}$ with the line that goes through $\boldsymbol{x_i}$ and $\boldsymbol{x_{i+1}}$ [Poy20], and

$$W^s = \frac{1}{2} k_s \sum_{e=1}^{n} \frac{(l_e - l_0)^2}{l_0^2} = \frac{1}{2} k_s \sum_{i=1}^{n} \frac{(||\boldsymbol{x_{i+1}} - \boldsymbol{x_i}|| - l_0)^2}{l_0^2} \tag{3.2}$$

where l_0 is the original length of a segment, l_e its current length and k_s the stretching constant. Figure 4 indicates the measure θ^i. Then, the total elastic energy is

$$W^{el} = W^b + W^s \tag{3.3}$$

As they are energy potentials, the elastic force will be minus its gradient multiplied, i.e.

$$\boldsymbol{g}^{el} = -\nabla_x W^{el} \tag{3.4}$$

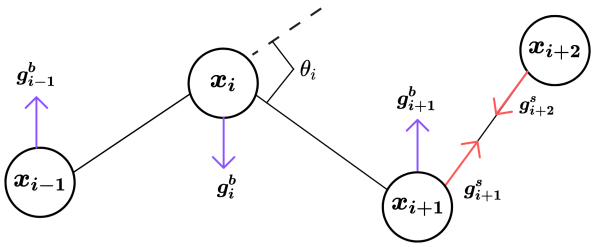


Figure 4: Scheme with geometrical description of a worm

Apart from these forces, the viscous and the internal (or motor) forces will be considered. The first ones model the viscous friction to which the worm is subjected to and, the second ones, the internal moments exerted by the worm in order to deform itself and move. The viscous forces are proportional to the velocity in which the worm is deforming and against it:

$$\boldsymbol{g}_i^{\eta} = -(\eta_t \boldsymbol{v_t} + \eta_n \boldsymbol{v_n}) \tag{3.5}$$

where $\boldsymbol{v}_t$ and $\boldsymbol{v}_n$ are the velocities along the tangential and normal directions with respect the body line. An important result that is proved on [D22] is that the center of mass remains still under

isotropic friction on flat surfes and incompressible materials. This is the case for the mathematical model being used here, and, consequently, a **non-isotropic** (with different values for the tangential η_t and the normal η_n components) viscous environment will be set.

The problem of finding the final position of a worm given an initial position and a set of moments, corresponds to finding the position where the forces are in **equilibrium**, as, in the case of having a non-zero net force, this would translate to an acceleration and, thus, to a movement. In consequence, the balance equations to solve are:

$$\boldsymbol{\eta \dot{x}_i} + \nabla_{x_i} W^{el}(\boldsymbol{x}) = \boldsymbol{g_i^m(M)} \tag{3.6}$$

where there is an unknown variable: either $\boldsymbol{M}$ is known and $\boldsymbol{x}$ is not (direct problem), or $\boldsymbol{x}$ is known a $\boldsymbol{M}$ is not (indirect problem). Note that $\boldsymbol{\dot{x}^i}$ will be approximated using the values of the positions. To solve this problem is equivalent to solve the next one

$$0 = \boldsymbol{g}_i^{el} + \boldsymbol{g}_i^{\eta} + \boldsymbol{g}_i^{m} \ , \ i = 1, ..., n+1 \tag{3.7}$$

which corresponds to finding the 0 of an equation. What the direct computation does is to find the final position $\boldsymbol{x}$ that solves equation 3.1 given an initial position, certain parameters and the internal moments $\boldsymbol{M}$.

3.2 Limitations of the model

In order to be able to properly build a Dataset for training, validating and testing our Neural Network, a preliminary analysis on the limitations of the mathematical model was conducted. More specifically, the point of interest was on determining values of bending stiffness k_θ that allowed the synthetic worm to bend enough to reproduce the experimental worm's movement.

In this situation, the values that were most important to compute were: M_{max} (the maximum amplitude of the moments accepted by the model) and C_{max}, corresponding to the maximum curvature observed applying a certain set of moments.

How are curvature and the C_{max} computed?

The curvature θ_i^k on the interior node i and at time-step k was initially computed as $\arccos(\frac{\boldsymbol{u}\cdot\boldsymbol{v}}{||\boldsymbol{u}||||\boldsymbol{v}||})$ where $\boldsymbol{u}$ is the vector going from node $i-1$ to node i and $\boldsymbol{v}$ the vector going from node i to node $i+1$. However, as this value clearly depended on the number of nodes n chosen to simulate the worm, it was decided to make it invariant by dividing this value by the length l of a segment. The max curvature is computed as $\max\ \theta_i^k\ \forall i, k$.

How is M_{max} determined?

To determine M_{max}, a program runs 100 steps of a sinusoidal movement that changes upon time. M_{max} is, then, the maximum value for M that the model can attain before convergence is lost.

$\boldsymbol{k_\theta = 0.01}$		
Values of η	$\boldsymbol{M_{max}}$	$\boldsymbol{C_{max}}$
$[0.07, 0.1]$	0.39	14.51
$[0.14, 0.2]$	0.4	14.02
$[0.1, 0.5]$	0.42	13.92
$[0.1, 2]$	0.48	13.89
$[0.02, 1]$	0.36	14.05

$k_\theta = 0.05$		
Values of η	M_{max}	C_{max}
$[0.07, 0.1]$	1.75	13.73
$[0.14, 0.2]$	1.75	14.23
$[0.1, 0.5]$	1.8	13.62
$[0.1, 2]$	1.75	11.49
$[0.02, 1]$	1.6	10.82

$k_\theta = 0.1$		
Values of η	M_{max}	C_{max}
$[0.07, 0.1]$	3.3	14.18
$[0.14, 0.2]$	3.25	13.30
$[0.1, 0.5]$	3.35	14.13
$[0.1, 2]$	3.25	11.13
$[0.02, 1]$	3.10	11.69

$k_\theta = 1$		
Values of η	M_{max}	C_{max}
$[0.07, 0.1]$	18.5	8.30
$[0.14, 0.2]$	18.4	8.58
$[0.1, 0.5]$	18.6	9.26
$[0.1, 2]$	18.9	9.40
$[0.02, 1]$	18.9	9.03

The main insights that can be extracted from the previous tables are that it seems that the maximum value for M, whilst it varies greatly based on k_θ, is pretty similar regardless of the values for η once k_θ is fixed. Also, for small values of the bending stiffness constant, here have been tried between 0.01 and 0.1, it seems that the curvatures the model is capable to achieve are pretty similar, around a value of 13. However, with higher values the achievable curvature is reduced, which makes sense because, at the end, k_θ indicates the bending stiffness. In summary, we have that $M_{max} \approx ck_\theta C_{max}$, with $c \in [2, 2.5]$.

When analysing the maximum curvature in the first 100 steps of a sample of experimental images, the value is set at 17.53, which is higher than the maximums computed here. This difference doesn't matter when fitting synthetic worms, but could be problematic when trying to fit experimental positions, as the model would not have seen this magnitudes of curvatures during training and, even finding the appropriate moments, the numerical implementation of the model might not converge.

4 Methodology

In this section, the particular implementation of the Machine Learning algorithms previously described will be discussed. It is important to remark that all the code created during this thesis has been written either in Matlab or Python. Considering that the programs created in previous projects were written in Matlab, the main code and the simulation of the mathematical model have been run in this language. However, given the versatility and relevance of Python in the field of Machine Learning, the Neural Networks have been coded in Python. In particular, the libraries of Pandas, TensorFlow, NumPy and Matplotlib have been used. All of this has been done using Anaconda and the Spyder environment.

4.1 The Idea

As it has been stated before, the main objective of the thesis was to validate whether a certain theory on worm's motion holds. The mathematical model that was developed according to it was implemented in Matlab and, to run, it needed, apart from the values of certain parameters, to be fed with a set of internal moments: one for each interior node (those which are nor the first nor the last one). However, it is not possible to measure directly those internal moments from experimental worms. What can be measured, though, are their successive positions, as it can be seen in figure 5.

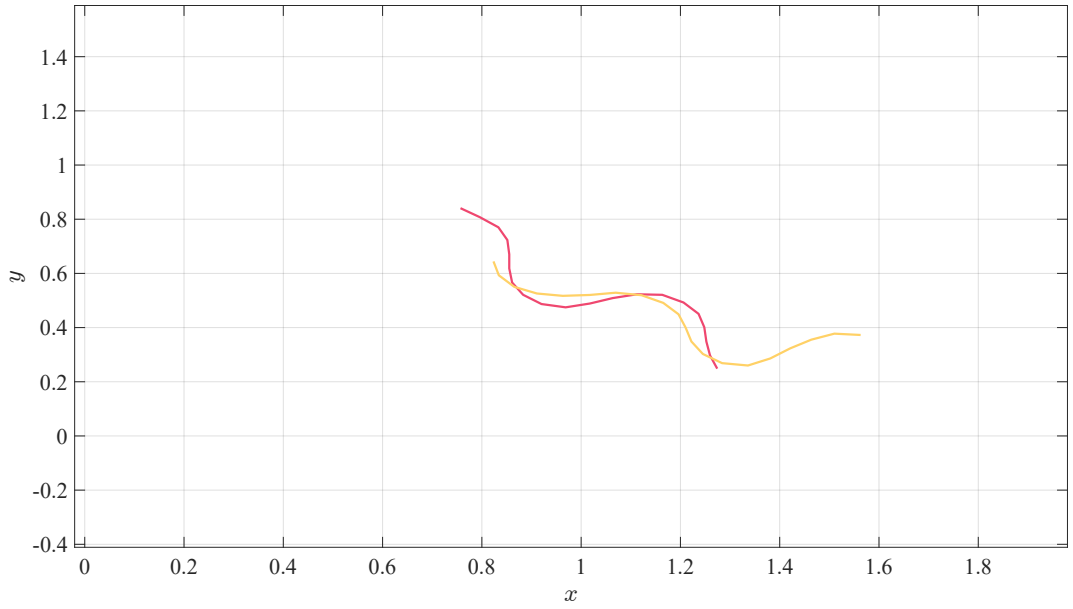


Figure 5: Example of experimental positions of a worm

At this point, is where Machine Learning algorithms come into play: they will be trained to induce moments from data, being that data only positions, displacements or the previously computed moments. That is, an approximation of the moments exerted by the worm will be computed indirectly.

One may logically wonder, then, how can those algorithms be trained if, as it was said before, it is not possible to measure the internal moments of a worm. The key, here, is that the objective is to check the validity of the existing mathematical model. Thus, the algorithms will be trained with synthetic data and then their performance will be measured on experimental data. That is, the already existing programs that simulate the mathematical model, will be fed (almost) random moments and an initial condition and the output will be a certain final position. Then, the algorithms will be trained using as input data the initial and final positions and their prediction will be the moments that caused this displacement. As the real (synthetic) moments are known, the algorithms will try to minimize a certain metric regarding the difference between the predicted and the real moments.

When the algorithms' accuracy is good enough, they will be used for validating the current mathematical model. To do so, these algorithms will induce internal moments from experimental data of a real worm [3] and, after, this moments will be fed to the mathematical model in order to check which path did the worm follow according to it. If the predicted and real paths were really similar, this would mean that the mathematical model was accurate. If they showed important differences, however, this would mean that the model was either incorrect or incomplete.

4.2 Implementation

4.2.1 Training, validation and test datasets

To be able to train a Neural Network and evaluate its results, it is fundamental to have something to train it with. In many cases this would imply to gather millions of inputs and their corresponding outputs from various sources, for instance, by collecting millions of images and their corresponding labels. However, due to the nature of this project, all the instances can be created autonomously by running many times the Matlab code simulating the mathematical model previously described and saving its inputs and outputs.

The difficulty, in this case, does not come from having to search for a lot of data on the internet or that an immense quantity of resources needs to be poured into the project in the form of people labeling data, but from finding the best way to represent all the possible movements that an experimental worm could perform. With this idea in mind, several ways of creating datasets were considered.

Creation of the datasets

In general terms, the creation of every instance in a dataset will follow these steps:

1. A random initial posititon will be computed, calculated by applying k consecutive random moments.
2. Having set this position as $\boldsymbol{X0}$ a new set of moments will be applied, and the initial position $\boldsymbol{u_{n-1}}$, the final one $\boldsymbol{u_n}$ and the moments $\boldsymbol{m_n}$ applied will be saved, as long as the numerical model had converged.
3. This will be repeated again s times (usually $s = 5$), reusing the previous final position as the new initial one.
4. After this, the process will be restarted: a new random initial position will be computed. This will we done n times.

It is important to note that many options to set random moments have been tried and that this process of dataset creation may combine them in different ways to cover a wider spectrum of options. The different types of moments investigated are explained in the next sections.

Random moments with restrictions

The first idea was to create the most random possible combination of moments by setting random moments on every node. As doing so might have created too big differences between the moments of two consecutive nodes, the moments were not completely random, but were chosen in a certain interval around the moment in the previous node and were limited to a maximum absolute value (to ensure convergence).

[3] Recall that, as they should have a high accuracy with the synthetic data, they are supposed perform good predictions of the internal moments underlying worms' displacements.

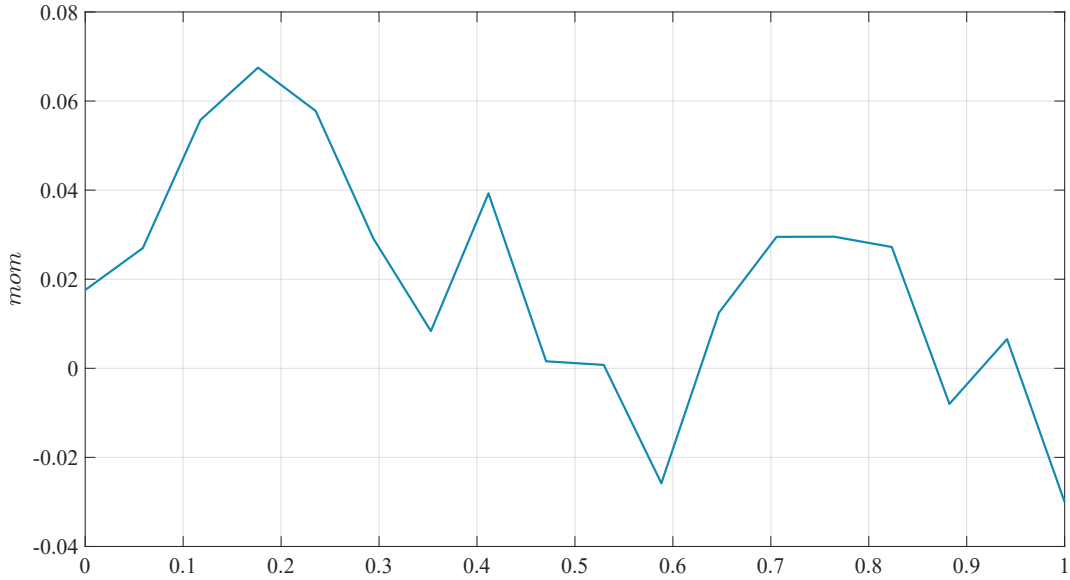


Figure 6: Example of random moments with restrictions

Splines

The main problem with the previous kind of moments was that, when trying to fit a simple synthetic trigonometric worm, the Neural Networks trained with that data didn't perform well. It was not until other options were used to compute the random initial positions, that the results became better. Because of this and to have smoother values, a new approach was taken: some random values would be chosen and, from those, a set of moments would be computed using spline interpolation.

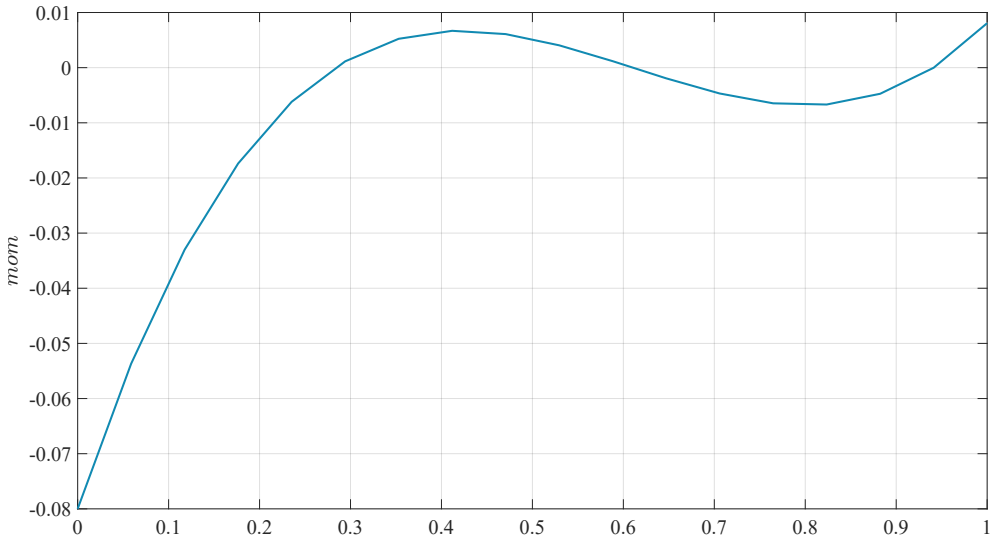


Figure 7: Example of moments interpolating using splines

Trigonometric moments

The results were not very good when fitting synthetic trigonometric worms, so it was decided to try training the Networks using the kind of moments that seemed to replicate the most the experimental worm's movement and that were being used to test the accuracy of the models: trigonometric moments.

In order to have a wider space of possible moments and getting inspiration by Fourier's solution

to approximate functions (Fourier Series) a better solution was found: using sums of trigonometric moments, which would return many more possible sets of moments than just having sines or cosines with different amplitudes, wave lengths and phases.

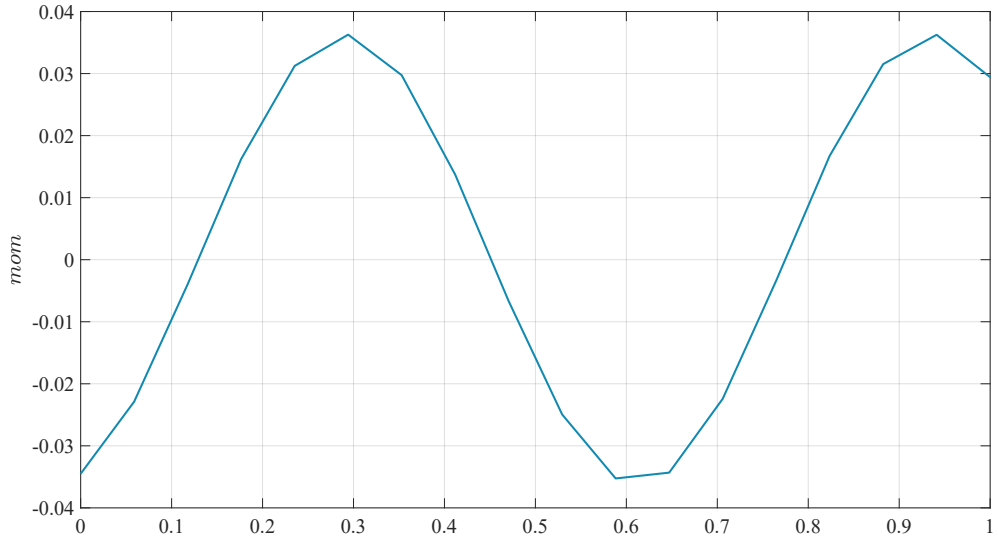


Figure 8: Example of sum of trigonometric moments

Partial moments, constant moments and moments with noise

Finally, with the objective of having more varied and representative datasets, three new kinds of moments were added:

1. **Partial moments:** Set of moments where only some nodes had non-zero moments.
2. **Constant moments:** All the moments having the same value.
3. **Moments with noise:** The moments are initially computed as a sum of trigonometric moments, as before, but a layer of noise is added to those values, by adding random values chosen from a normal distribution.

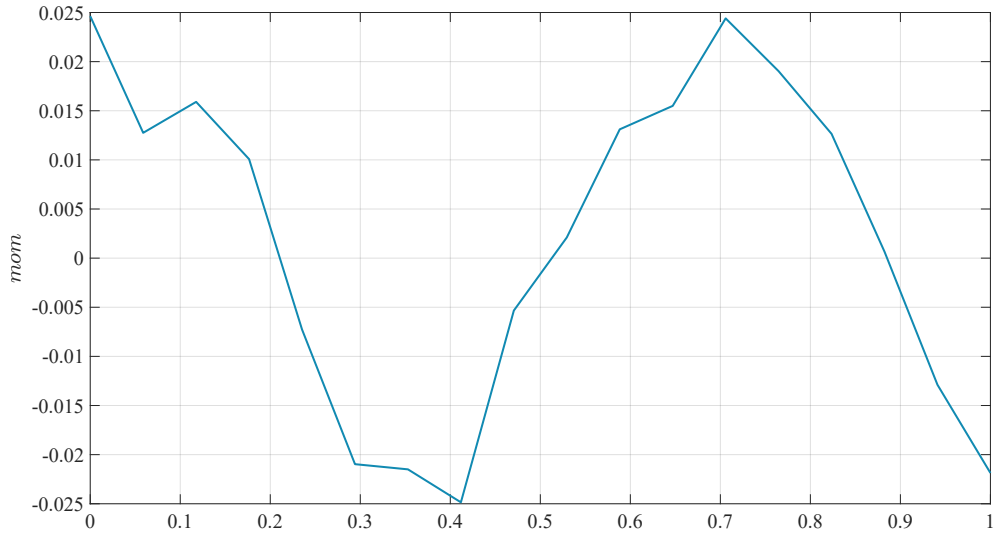


Figure 9: Example of sum of trigonometric moments with noise

4.2.2 Types of inputs

After determining the best ways to create the training datasets, the parameters that will be stored and later used as Input and Output for the Neural Networks ought still to be chosen. Those could be the positions of the worm, displacements, the moments exerted and many more. To be able to understand properly the different options that were tried, the symbols used are explained on table 1.

Symbol	Meaning
$\boldsymbol{u_n^{exp}}$	Experimental position of the worm at time step n.
$\boldsymbol{u_n^{ML}}$	Position predicted by the Machine Learning algorithm at step n.
$\boldsymbol{\Delta u^{ML}}$	Displacement respect prediction: $\boldsymbol{u_{n+1}^{exp} - u_n^{ML}}$.
$\boldsymbol{\Delta u^{exp}}$	Displacement respect experimental data: $\boldsymbol{u_{n+1}^{exp} - u_n^{exp}}$.
$\boldsymbol{m_n}$	Moment at step n.
$\boldsymbol{m_n^{ML}}$	Predicted moment at step n.

Table 1: Table of symbols

Essentially three options for the input were considered:

1. **NN1:** Taking as input only the final position $\boldsymbol{u_n^{exp}}$ of the worm. With this approach, the result is quite robust, as it only depends on the time-step n. However, it doesn't take into account the fact that the movement is path dependent, as, due to friction, the initial position is very important.
2. **NN2:** To solve this issue, the solution is to provide both $\boldsymbol{u_{n-1}^{exp}}$ and $\boldsymbol{u_n^{exp}}$. With this, what counts is not only the displacement performed, but also the position where it started. This formulation would be very similar to providing $\boldsymbol{\Delta u^{exp}}$ and $\boldsymbol{u_{n-1}^{exp}}$, which, in fact, is the option that was finally used.
3. **NN3:** The last proposal of input would consist on inputting $\boldsymbol{u_{n-1}^{exp}}$, $\boldsymbol{\Delta u^{exp}}$ and the previously predicted moment $\boldsymbol{m_{n-1}^{ML}}$. In this case, the results end up being quite similar, while the model becomes more complex. It must also be noted that this added information could easily become problematic, as relying too much on the previously computed moments would case the errors in the predicted moments to propagate rapidly.

A summary of this can be found on table 2.

4.2.3 Rigid body motion

In order to ease the job to the Neural Network, some prior computations are done. The objective of these is to always feed the algorithm with worm's positions which are centered in the origin and whose regression line is the x axis. This is important because, then, the training data for the models will be much easier to create and, in fact, it will be more complete. If no prior computations were applied, the range of data that the algorithms might receive would be too big to make any sense out of it and it would not be very easy to create a training set covering all the options. However, as translations and rotations commute with the application of moments, we can restrict ourselves to the centered and rotated case.

Name	Input	Output
NN1	$\boldsymbol{u_n^{exp}}$	$\boldsymbol{m_n^{ML}}$
NN2	$\boldsymbol{u_{n-1}^{exp}}$, $\boldsymbol{\Delta u^{exp}}$	$\boldsymbol{m_n^{ML}}$
NN3	$\boldsymbol{u_{n-1}^{exp}}$, $\boldsymbol{\Delta u^{exp}}$ and $\boldsymbol{m_{n-1}^{ML}}$	$\boldsymbol{m_n^{ML}}$

Table 2: Types of input

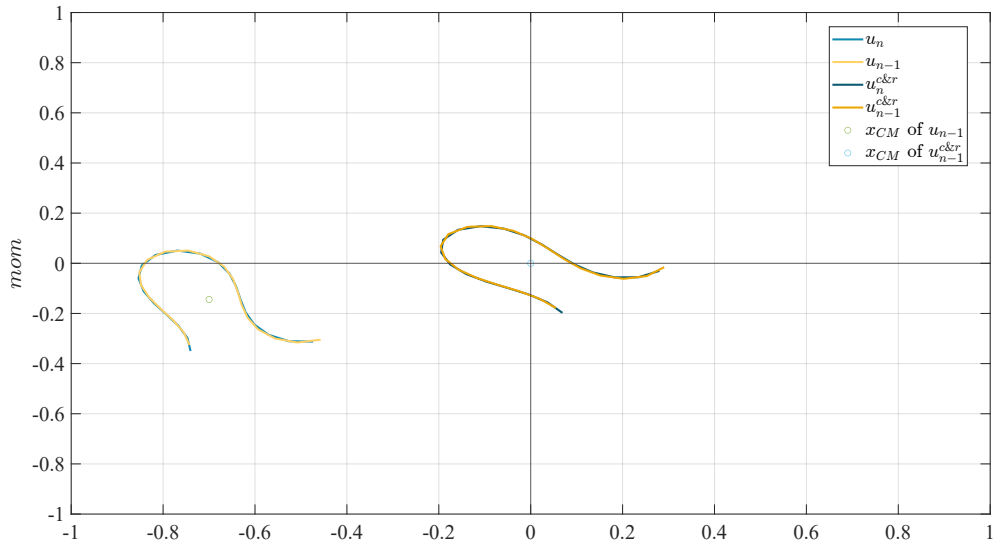
These prior computations will start by computing the rigid body movement of the worm. That is, the objective is to determine both the translation and the rotation performed by the worm from an imaginary original resting position. To detect the first, the center of masses (CM) of the worm will be determined and, then, subtracted to all the positions of the worm's nodes, transferring it to the origin.

$$\boldsymbol{u_n^c} = \boldsymbol{u}_n - \boldsymbol{x_{CM}} \tag{4.1}$$

For the second, the regression line of the new positions of these nodes, the line that bestly fits those positions, will be found. Then, as this line will pass through the origin (it passes through the CM), its slope θ will indicate the rotation of the rigid body movement and the positions of the worm will be rotated $-\theta$ radians. $\boldsymbol{R}_n$, here, is the rotation matrix corresponding to a rotation of $-\theta$ radians.

$$\boldsymbol{u_n^{c\&r}} = \boldsymbol{R}_n \boldsymbol{u_n^c} \tag{4.2}$$

These prior computations are calculated from the previous position of the worm u_{n-1} but they will be applied both to u_{n-1} and u_n. This is because we want to move both positions together so that the relative positions of the two (and, thus, the displacement caused by the moments) remain the same.



4.2.4 The Neural Network

The neural network will be used to compute the moments exerted by the worm given a certain input. As described before, the option that ended up working the best was to use $\boldsymbol{u_{n-1}}$ and $\boldsymbol{\Delta u}$. In both cases and regardless of the architecture chosen, the process involving the neural network can be divided in two parts.

Training

Once chosen and created the Dataset that will be used, the model or models can start being trained. To do so, the data has to be processed to have the appropriate dimensions and separated into input and output. For all cases, the output will be the moments $\boldsymbol{m_n}$ exerted by the synthetic worm. After this, the data has to be partitioned into three subdatasets: the training, the validation and the test datasets. As their names indicate, the first will be used for training purposes, the second to validate that the model is not overfitting too much and the last for evaluating the results on a sample of data that the model has not seen before.

Now, it is necessary to set the architecture of the model, by adding the different layers that it will be composed of, and to establish some parameters that will determine the way in which the model is trained: the number of epochs, the learning rate (or the optimizer), the batch size and the loss function. The first corresponds to the number of times that the training data will be visited to update the parameters, the second to how much will the model update in every iteration, the third to the number of samples analysed before any update and, the last, to the function that will determine the distance to the desired output.

Execution

When executing the model, the only necessary thing will be to feed the network with the same kind of Input it was trained on and to apply the moments that come out from it. That is, if the model was trained using $[\boldsymbol{u_{n-1}}, \boldsymbol{\Delta u}]$ as input, when calling it, for the output to make sense, the inputs should also be of the form $[\boldsymbol{u_{n-1}}, \boldsymbol{\Delta u}]$ and not of the form $[\boldsymbol{u_{n-1}}, \boldsymbol{u_n}]$, for instance.

Limitations of the model

Due to the limitations of the model observed in section 3.2, the decision was taken to restrict the output moments to the **admissible domain**. More specifically, what is done is to set output values higher than M_{max} or smaller than $-M_{max}$ to M_{max} and $-M_{max}$, respectively. That is, all moments outside of the admissible domain are projected to the nearest boundary value.

5 Results

In this section the results of the project will be shown and discussed. In a first part, the three proposals of Input described in the Methodology section (NN1, NN2 and NN3) will be compared. In a second, the results of creating the training dataset using different techniques to generate the random moments will be discussed. Then, two fundamental elements related to the problem will be mentioned: the sensibility near the limits of the admissible domain and, more importantly, the relevance of the parameter $\boldsymbol{\eta}$. After that, a comparison using different architectures will be done. Finally, and using the best model found will be used to to fit both randomly created synthetic worms and the interpolated images of experimental worms[4].

5.1 Comparison of Inputs

In order for this comparison to make sense, the dataset that was used to train the three models is the same. It has 50000 samples of data and was created using sums of trigonometric moments to which some random noise was added. In some cases, also, partial moments were applied, in the sense that only a segment of the moments would change from one step to the next one, remaining the other values fixed. It is also important to note that, regardless of the model trained, the architecture was always the same, with a hidden dense layer having 128 units and an output layer of size 18. Then, the only things that changed would be the input layer and the formet of the data it received as input.

Definition 5.1 (Mean squared error)**.** *The mean squared error is a metric used to assess the performance of predictors and estimators. Let n be the number of datapoints of a sample, $\boldsymbol{m}$ the matrix of values of our target variables and $\boldsymbol{m^{pred}}$ the matrix of predicted values. Then*

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{m_i} - \boldsymbol{m_i^{pred}})^2 \tag{5.1}$$

Definition 5.2 (Error in predicted moments)**.** *Let m be the number of nodes used to interpolate a worm, M the value of the real moments and M^{pred} the prediction made by a model. Then, the error metric used to assess the performance of the models by comparing their prediction to the real values of the moments is computed as*

$$EPM = \frac{1}{m}(\sum_{j=1}^{m}|\boldsymbol{M_j} - \boldsymbol{M_j^{pred}}|)(\frac{1}{\frac{1}{m}\sum_{j=1}^{m}|\boldsymbol{M_j}|}) = \frac{\sum_{j=1}^{m}|\boldsymbol{M_j} - \boldsymbol{M_j^{pred}}|}{\sum_{j=1}^{m}|\boldsymbol{M_j}|} \tag{5.2}$$

Definition 5.3 (Error in predicted positions)**.** *Let m be the number of nodes used to interpolate a worm, $\boldsymbol{u}$ the value of the real positions, $\boldsymbol{u^{pred}}$ the prediction made by a model and l the length of the worm. Then, the error metric used to assess the performance of the models by comparing their prediction to the real values of the positions is computed as*

$$EPP = \frac{1}{ml}\sum_{j=1}^{m}||\boldsymbol{u_j} - \boldsymbol{u_j^{pred}}|| \tag{5.3}$$

Here, $\boldsymbol{u_j}$ and $\boldsymbol{u_j^{pred}}$ indicate the displacements at node j.

Two kind of metrics will be used to determine the performance of the three options. On the one hand, the mean squared error of the models in the test set. On the other, two metrics were used to determine the performance when trying to fit a randomly generated synthetic worm, a first one measuring the error between the predicted and the synthetic moments and a second measuring the error between positions.

[4]The experimental data was provided by researchers at ICFO.

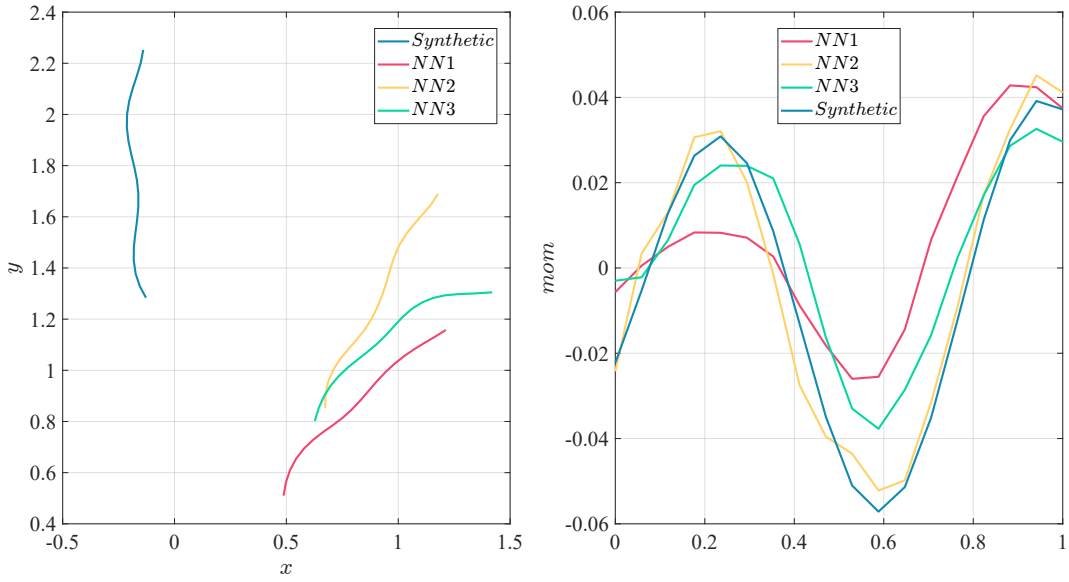
Type	MSE
NN1	8.68 x 10^{-5}
NN2	2.57 x 10^{-5}
NN3	2.16 x 10^{-5}

Table 3: MSE per Input type

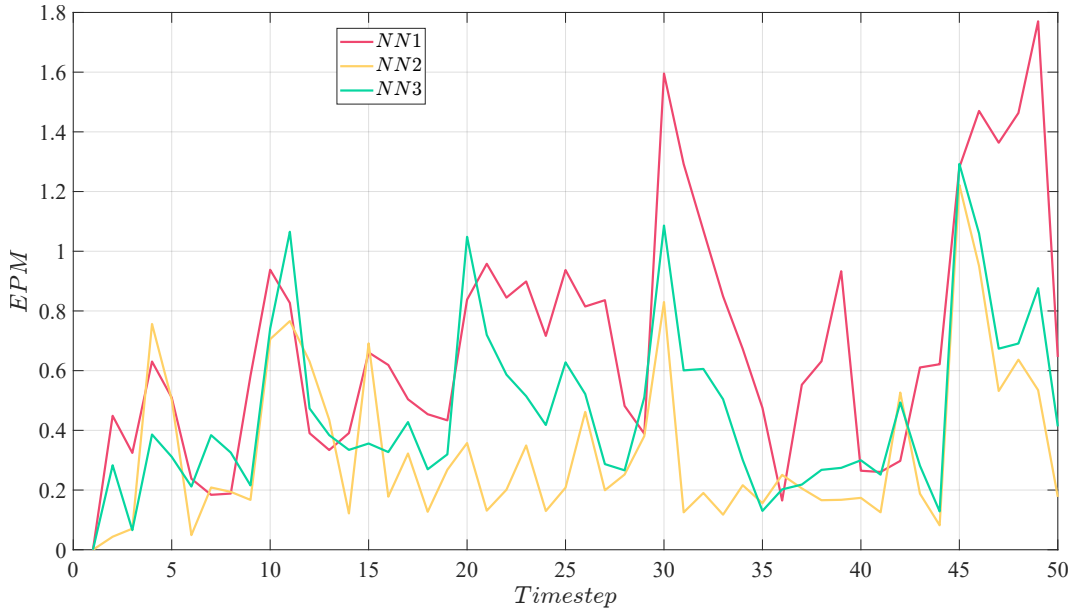
5.1.1 Mean squared error

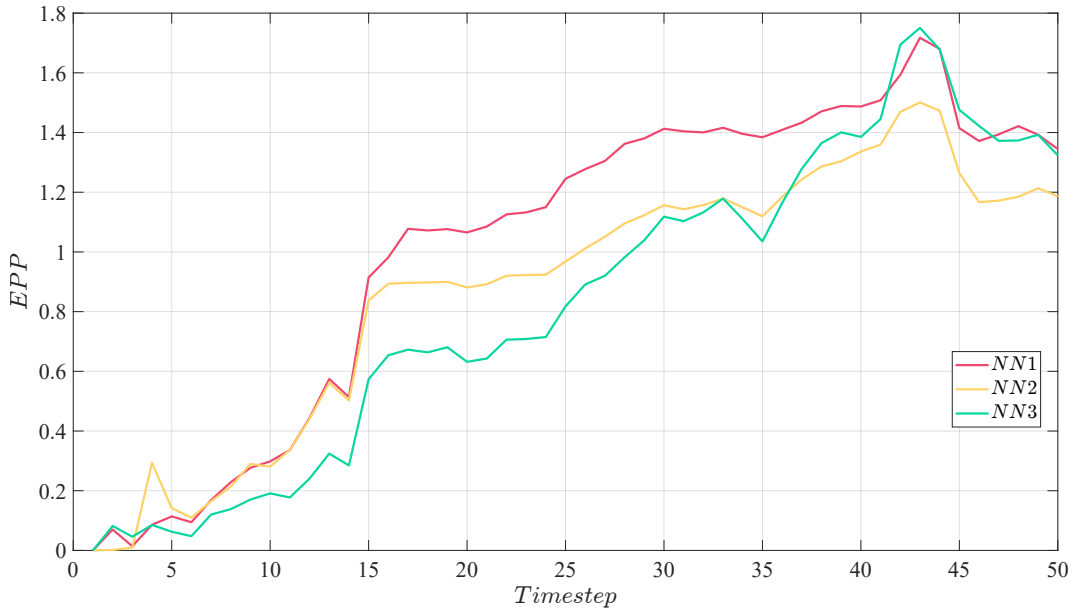
The values of the mean squared error after training for 100 epochs can be seen in table 3.

5.1.2 Snapshot after n = 50 steps



5.1.3 Comparison





The results of this analysis clearly show that one of the options is much worse than the others, which show a similar behaviour. As it has already been discussed, the option of NN1 has too little information to function properly. However, an strange behaviour happens with NN2 and NN3. Despite NN3 having more information on the movement of the worm, as it also includes the previously deduced moments, its errors are pretty similar, and, in fact, almost identical to those of NN2.

The most logical explanation to this phenomenon is that the model at NN3 doesn't take $\boldsymbol{m_{n-1}^{ML}}$ too much into account, and operates, de facto, as if it had the same inputs as NN2. This makes sense, as taking into account previously computed moments could be detrimental if those already had an error: the error would propagate and the new prediction would also be incorrect. Thus, relying only on $\boldsymbol{u_{n-1}}$ and on $\boldsymbol{\Delta u}$ is much more robust. All of this, combined with the fact that the model of NN2 would have less parameters than that one for NN3, situates NN2 as the best of the

Type	MSE
RMR	2.08 x 10^{-4}
Splines	4.92 x 10^{-6}
Trigonometric	3.48 x 10^{-5}

Table 4: MSE per moment creation technique

analysed options.

There is still though, a common problematic that is worth mentioning. Although the error of the moments for NN2 and NN3 have not been excessively large in general, except for two spikes, the distance of the positions ends up being quite remarkable. The main element causing this is the fact that this problem is very sensitive to small errors, specially when those happen near the limit of the admissible domain. That is, starting from the same position and applying two very similar sets of moments that are near the limit can cause the final positions to be very different. This problematic, as it will be discussed in section 6, should be taken into account in any future work.

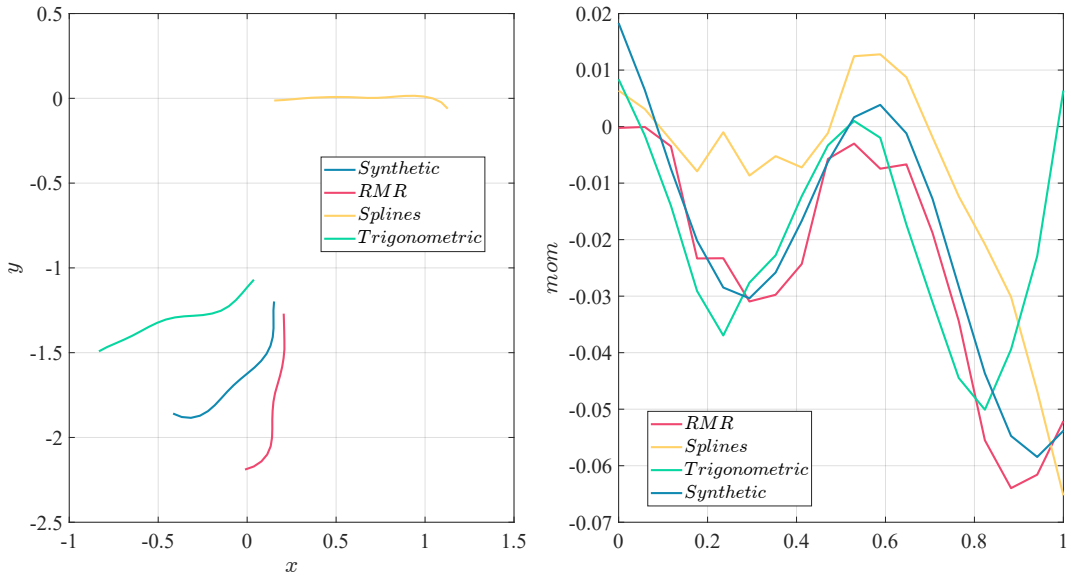
5.2 Comparison of moment creation techniques

In order for this comparison to make sense, the model trained with the three datasets, and their size, were the same. Its chosen input was NN2, as it showed to be the option that performed the best. Each dataset had 10000 samples.

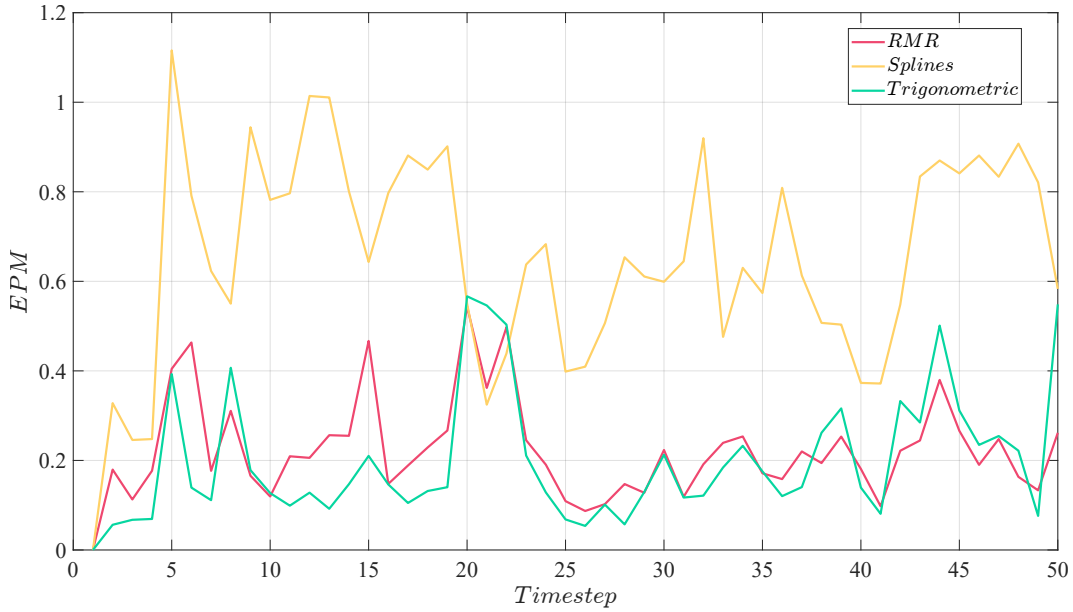
5.2.1 Mean squared error

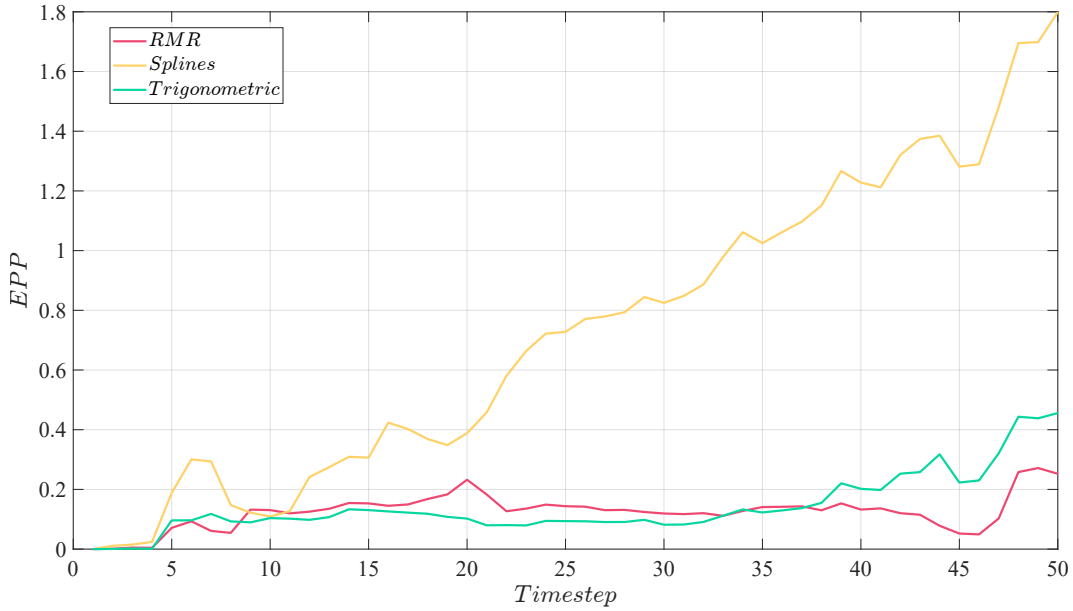
The values of the mean squared error after training for 100 epochs can be seen in table 4.

5.2.2 Snapshot after n = 50 steps



5.2.3 Comparison





As expected, the model trained using sums of trigonometric waves with some noise is the one working the best. Surprisingly, however, the option of random moments with restrictions also worked pretty well. This is probably because, in all cases, the moments were applied to deformed initial positions.

Despite of this, the sum of trigonometric moments with noise performed better and, more importantly, simplifies much more the creation of random moments that evolve on time, as in order to create them the only values to set are those of certain parameters, such as ω and *phase* that can be reused along several iterations. In fact, to create a random sinusoidal movement, the only necessary thing is to determine ω, *phase* and c

$$m_i = csin(\omega(x_i + \Delta t) + phase)$$

Doing this s times and adding the resulting values would give a sum of s trigonometric waves.

Then, the moments would be the evaluation of the resulting function in 18 equally spaced points, which would change on time.

5.3 Sensibility near the limits of the admissible domain

As it has already been mentioned, one of the limitations of the mathematical model is that some boundaries existed on the amplitude of the internal moments that the worm could be exerting. What was also noted during different stages of the thesis was that, near these boundary values, the deformation of the worm could change a lot even when the differences between moments were quite small. This phenomenon can be clearly observed in the two images of figure 10

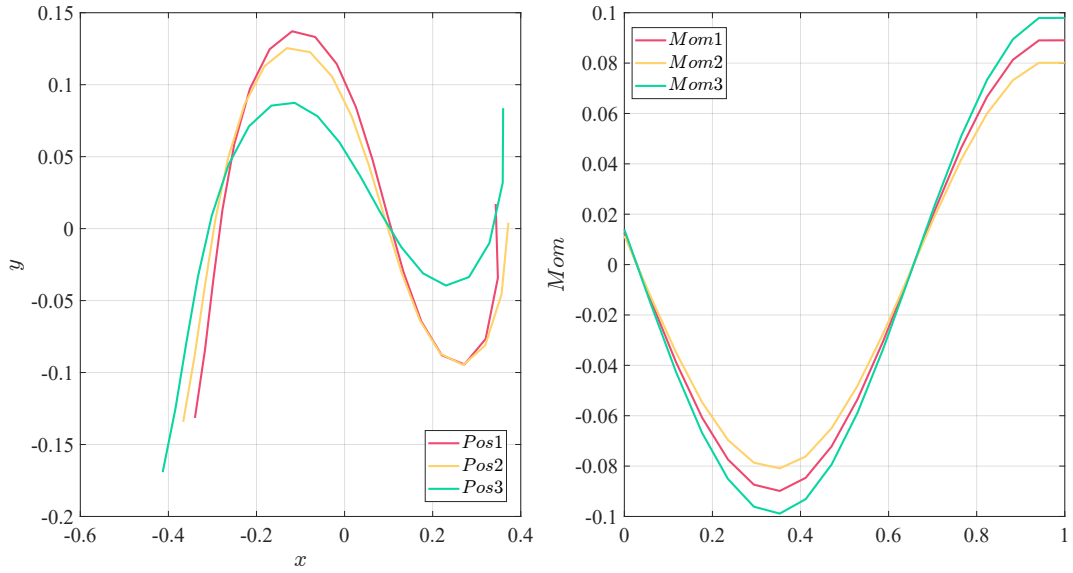


Figure 10: Plots showing the sensitivity near the limits of the admissible domain

These plots correspond to applying 5 consecutive times the following trigonometric moments:

1. $Mom1 = M * sin(5\boldsymbol{x} + 3) * 0.9$
2. $Mom2 = M * sin(5\boldsymbol{x} + 3)$
3. $Mom3 = M * sin(5\boldsymbol{x} + 3) * 1.1$

where $M = 0.09$. This indicates that it can be very important to train the models with a lot of data with high values, so that the parameters are set in such a way that they minimize the errors when predicting in this area.

5.4 Relevance of η

Along all the thesis, the relevance of the parameter (actually, parameters) $\boldsymbol{\eta}$ in the worm's dynamics has appeared to be fundamental, which should be taken into account to improve the models fitting of experimental data. As it can be observed in figures 11 and 12, each one showing the path transited by a synthetic worm applying the same moments and just changing the values of $\boldsymbol{\eta}$, the essence of the dynamics are completely different.

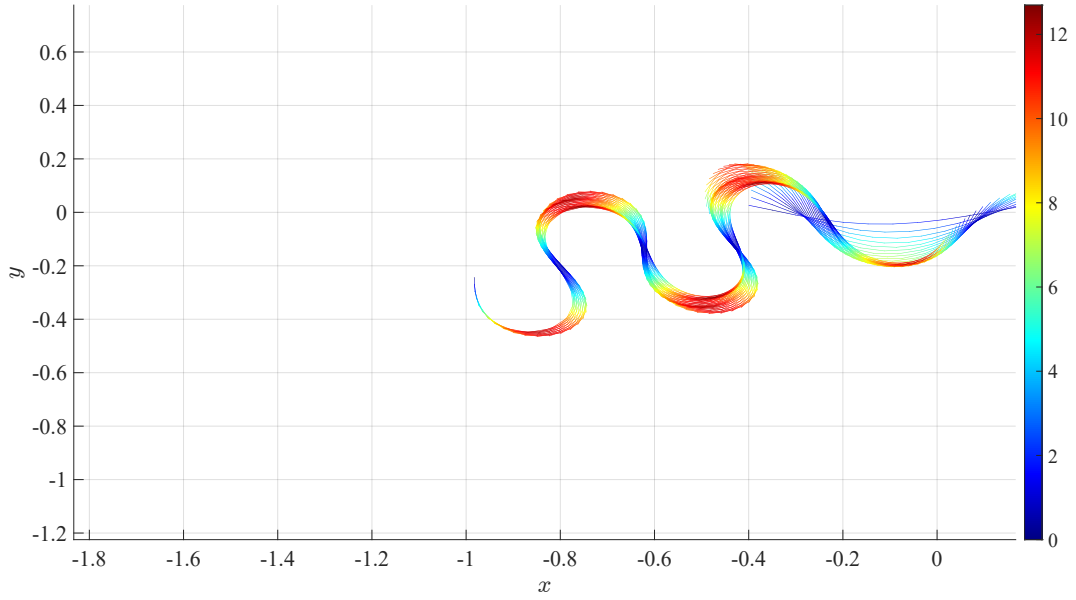


Figure 11: Curvature and path followed by a synthetic worm with $\boldsymbol{\eta} = [0.01, 1]$

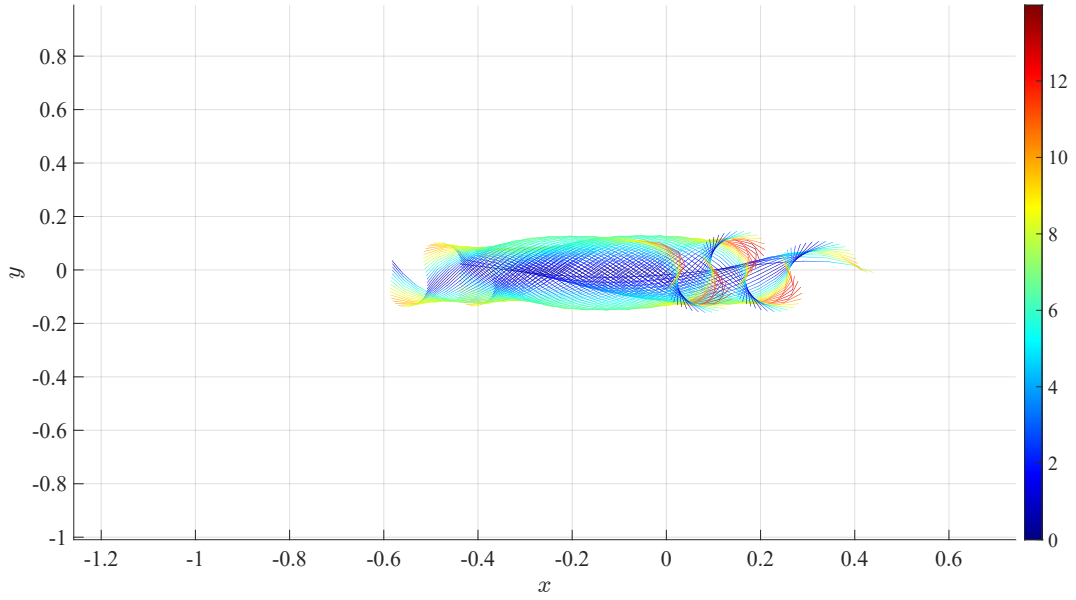


Figure 12: Curvature and path followed by a synthetic worm with $\boldsymbol{\eta} = [1, 2]$

While the first one seems to follow its own path, the second seems to sweep the floor. This shows that if one wants to create a training dataset that correctly represents what is actually happening, one should try to estimate the vales of $\boldsymbol{\eta}$ in one way or another. What has been observed during this project is that real worms seem to follow their own path as well, so the most likely proportions would be similar to the ones in the first figure. Figure 13 shows the path left by an experimental worm after 100 steps.

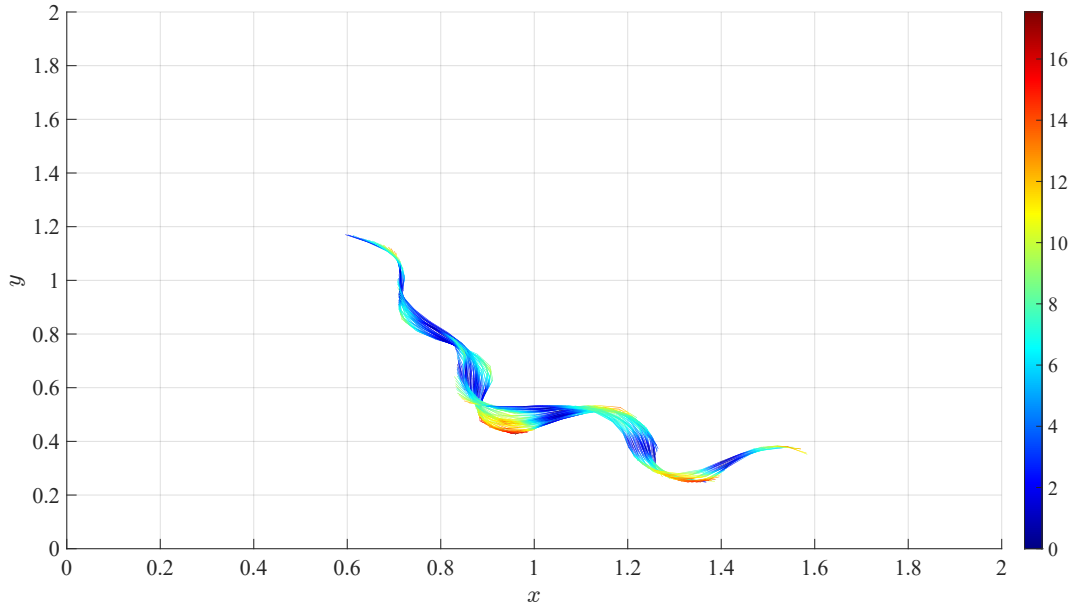


Figure 13: Curvature and path followed by an experimental worm after 100 steps

5.5 Comparison of Layer types

5.5.1 Mean squared error

The values of the mean squared error after training for 100 epochs can be seen in table **??**.

Type	**MSE**	**# of parameters**
Dense	5.87 x 10^{-5}	1881
CNN	1.44 x 10^{-4}	12690
RNN	3.77 x 10^{-5}	756

Table 5: MSE per layer type

5.5.2 Architectures of the models

Type	**Description**
Dense	Model containing three layers: an input layer with 80 units, a hidden dense layer with 128 and an output layer of 18 units.
CNN	Model containing four layers: an input layer with 80 units, a convolutional layer with kernel size 12 and 1 filter, a dense layer with 50 units and an output layer of 18 units.
RNN	Model containing three layers: an input layer with 80 units, a recurrent layer that iterates on the nodes of the worm and an output layer of 18 units.

Table 6: Architectures of the models

5.5.3 Snapshot after n = 50 steps

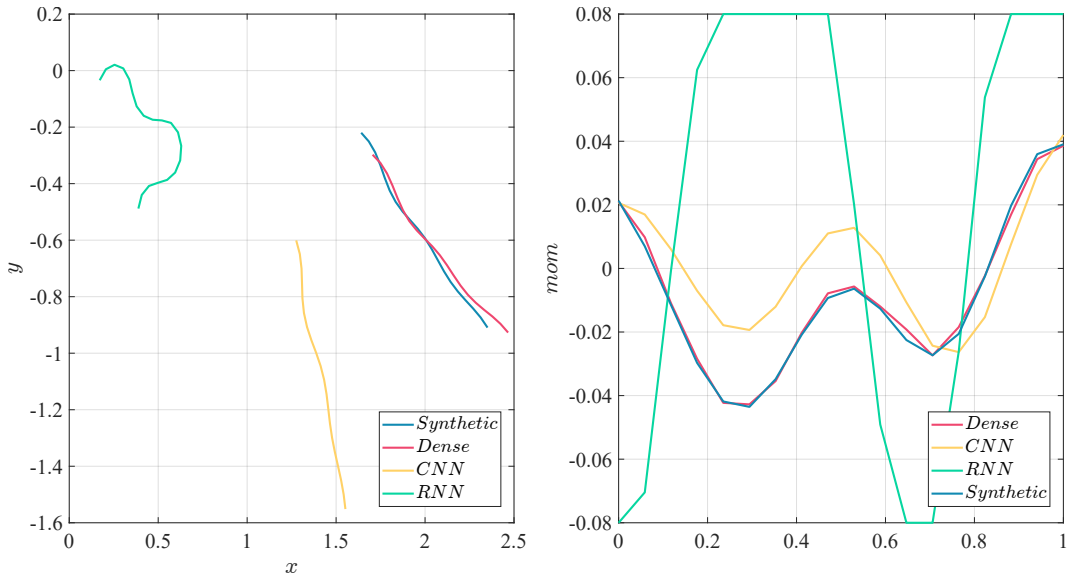


Figure 14: Positions and moments after n = 50 steps

5.5.4 Comparison

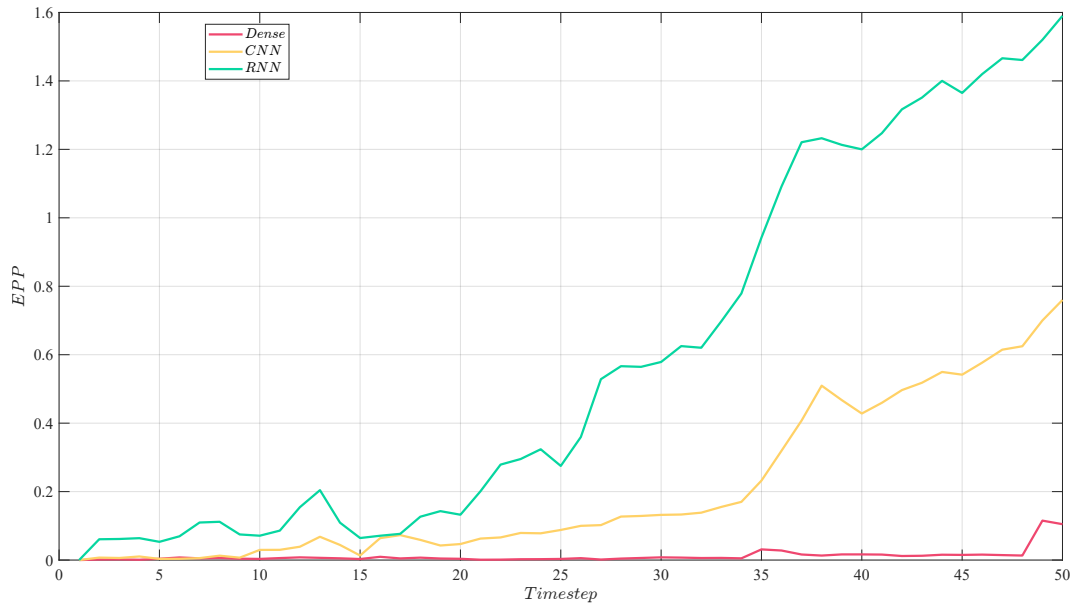


Figure 15: Evolution of the error in predicted positions

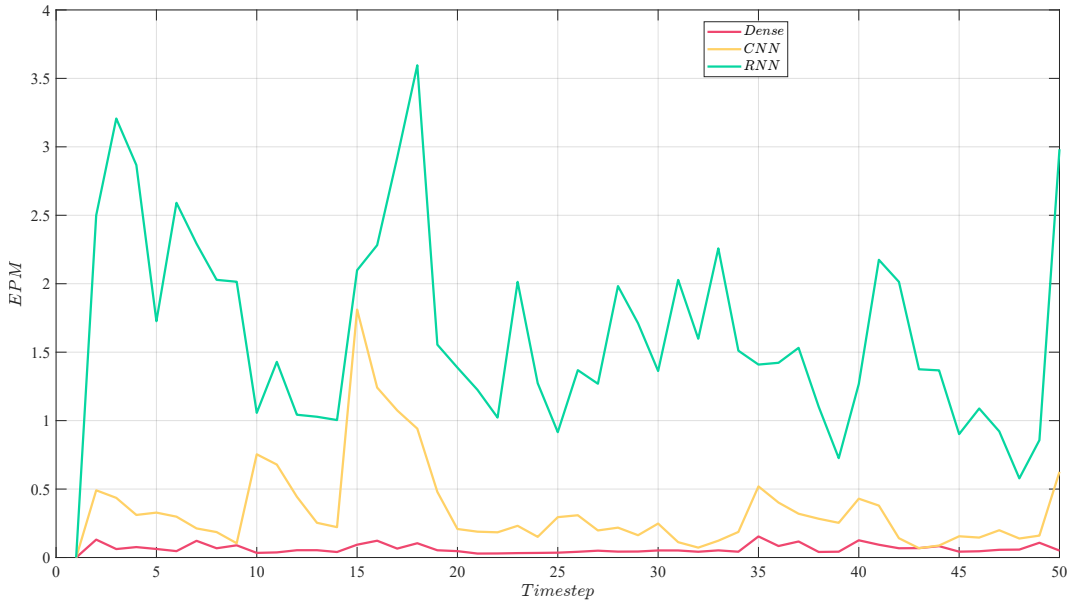


Figure 16: Evolution of the error in predicted moments

In theory, the options of using Convolutional and Recurrent layers seemed to be very good options to improve the accuracy shown by dense layers because they would add either translation invariance (CNN) or iterate various time on the input until getting a better prediction (RNN). The results shown in figures 15 and 16 indicate, however, that using just dense layers worked better, giving the option of using a recurrent layer the worst results. Even though using a convolutional layer does not perform as badly, the difference with using just dense layers is still big, as its error in moments is always at least 2 times bigger. For this problem, then, it seems that using all the available information at once can be better than processing it separately.

5.6 Final results

In this section, the results of fitting some randomly generated synthetic worms and experimental data with the best of the models tried and having trained it with the dataset that has the best results will be discussed. The values of $\boldsymbol{\eta}$ used both for the training and the evaluation stage are $[0.01, 1]$.

5.6.1 Synthetic tests

To test the models' performances in fitting synthetic worms, the models will be subjected to three tests. The first will check how capable is the model to detect moments that rapidly change, the second its capacity to follow a moving sum of sinusoidal waves and, the third, if it can capture constant moments. The model was trained on 150000 samples of data generated using sum of trigonometric moments with noise and trained for 1000 epochs.

Rhythm change

In this test, 50 consecutive sets of moments are applied, starting from an initial resting position centered in the origin. Every 5 steps, a new sum of sinusoidal functions is computed, which will be translated in time during the next 4 steps, when it will change again.

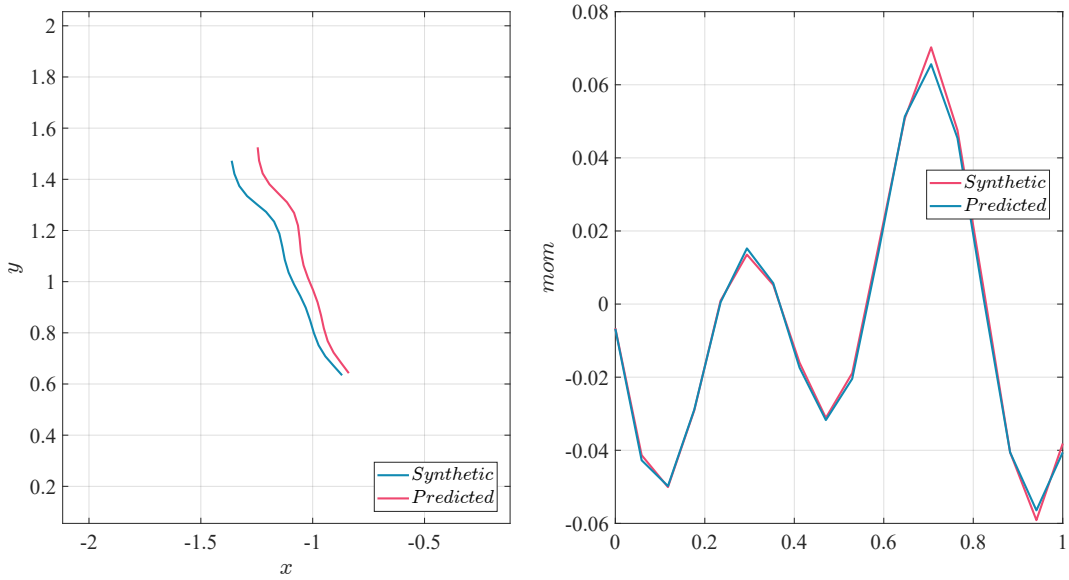


Figure 17: Positions and moments after n = 50 steps

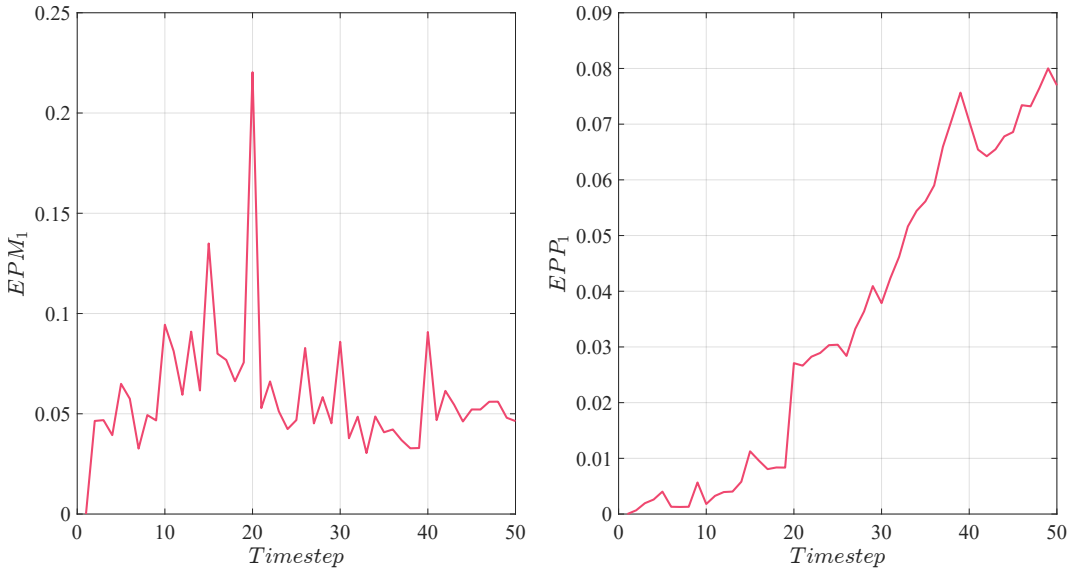


Figure 18: Errors corresponding to fitting in test 1

A spike on the graph monitoring the error in the prediction of moments can be found, coinciding with one of the rythm changes, as the displacements happening then can be more irregular. Still, the network does a pretty good job. The error in positions is not that high, but could still be improved by achieving a higher accuracy near the boundaries of the admissible domain and by projecting the predicted worm to the synthetic one every time that the distance passed a certain threshold.

Fixed sum of trigonometric waves

In this test, 50 consecutive sets of moments are applied, starting from an initial resting position centered in the origin. A sum of sinusoidal functions is computed at the beginning, which will determine the moments and will be translated on time.

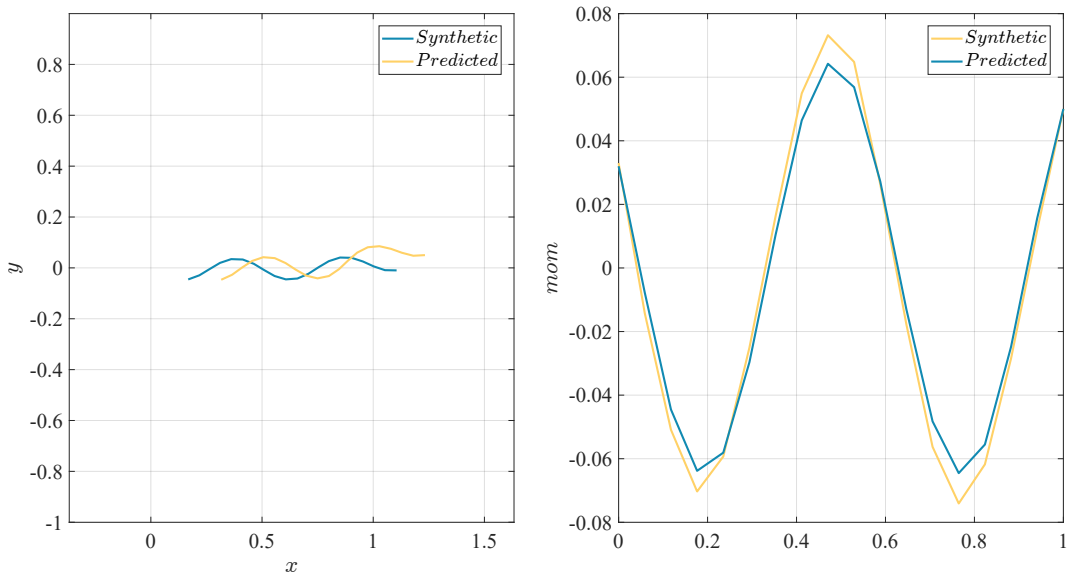


Figure 19: Positions and moments after n = 50 steps

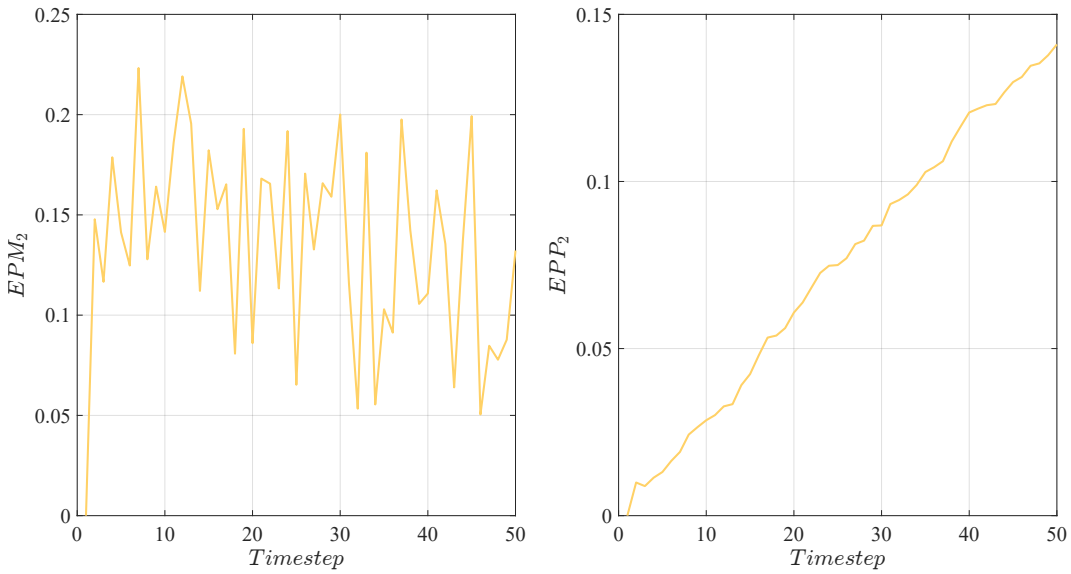


Figure 20: Errors corresponding to fitting in test 2

This test is probably more realistic as sudden and abrupt rythm changes are probably not that common in worm-like motion. As it can be observed, the error in the moments doesn't show spikes as before, but still shows some irregularities. The final positions are quite near though there is still some error, that grows steadily on time.

Constant moments

In this test, 10 consecutive sets of moments are applied, starting from a initial resting position centered in the origin. A set of constant moments is randomly chosen at the beginning and the same moments applied during the next 10 steps.

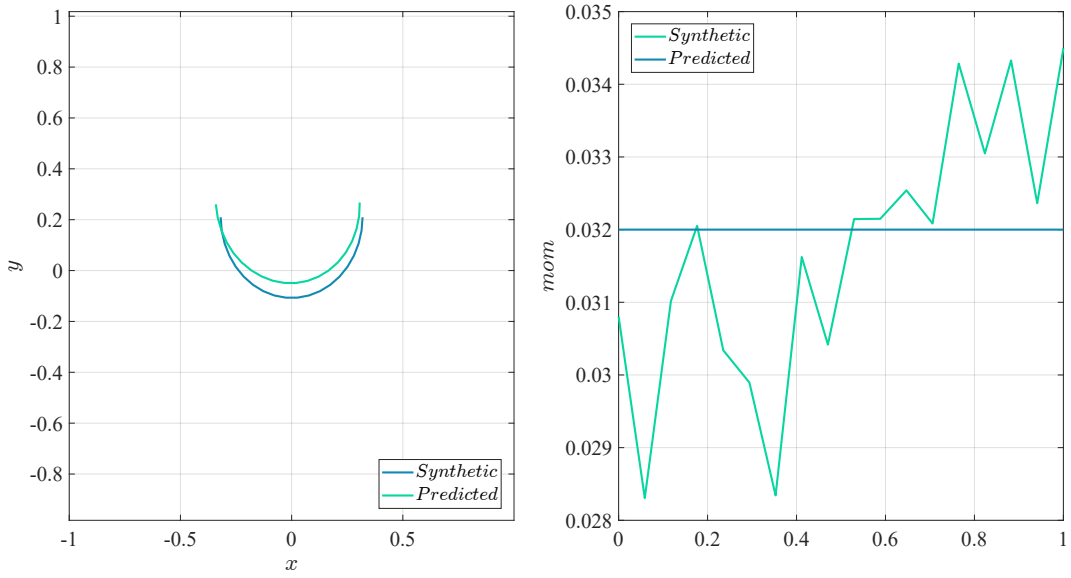


Figure 21: Positions and moments after n = 50 steps

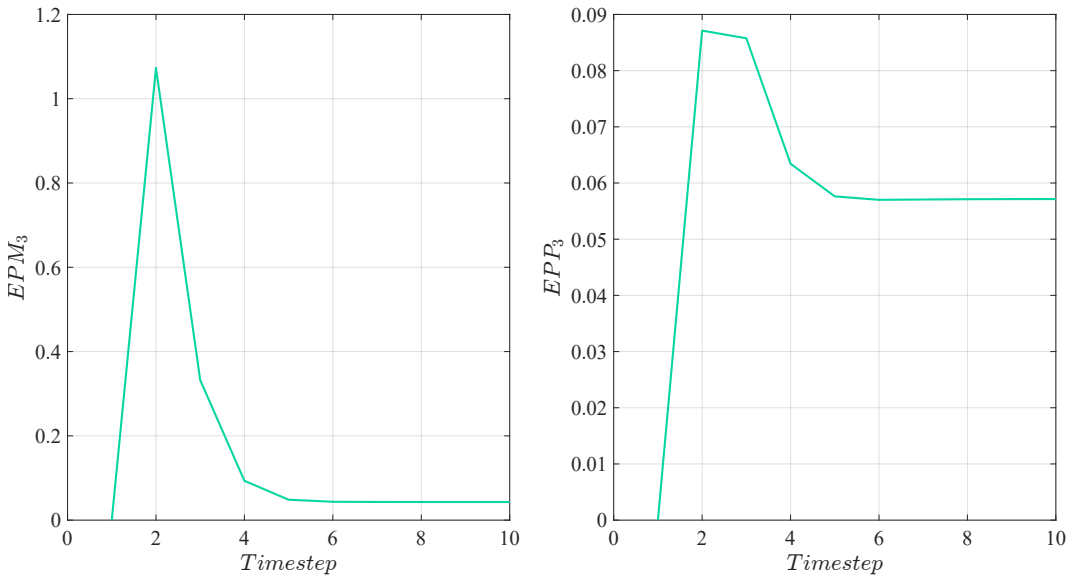


Figure 22: Errors corresponding to fitting in test 3

Despite not being perfect, the model does a pretty good job at getting to the final expected position, regardless of not having been trained on data having used constant moments. These are good news, as they indicate that the model is able to generalize outside of its training domain, at least to a certain extent. As the error plot for moments shows, in any case, work still needs to be done to achieve an optimal result.

5.6.2 Experimental tests

The same model used for the synthetic tests was tested with 100 consecutive positions of an experimental worm.

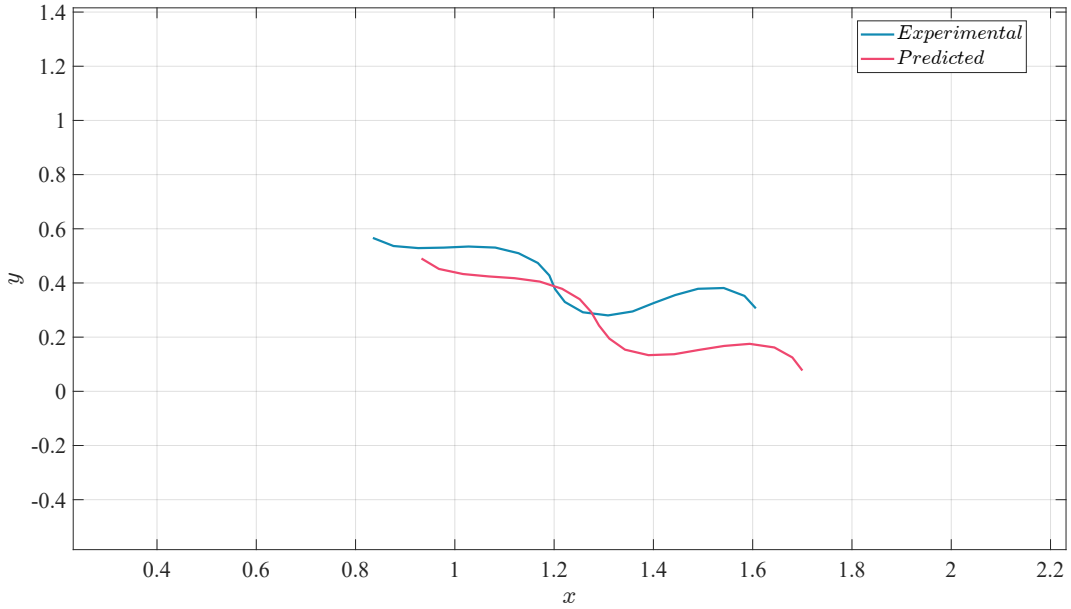


Figure 23: Positions and moments after n = 100 steps

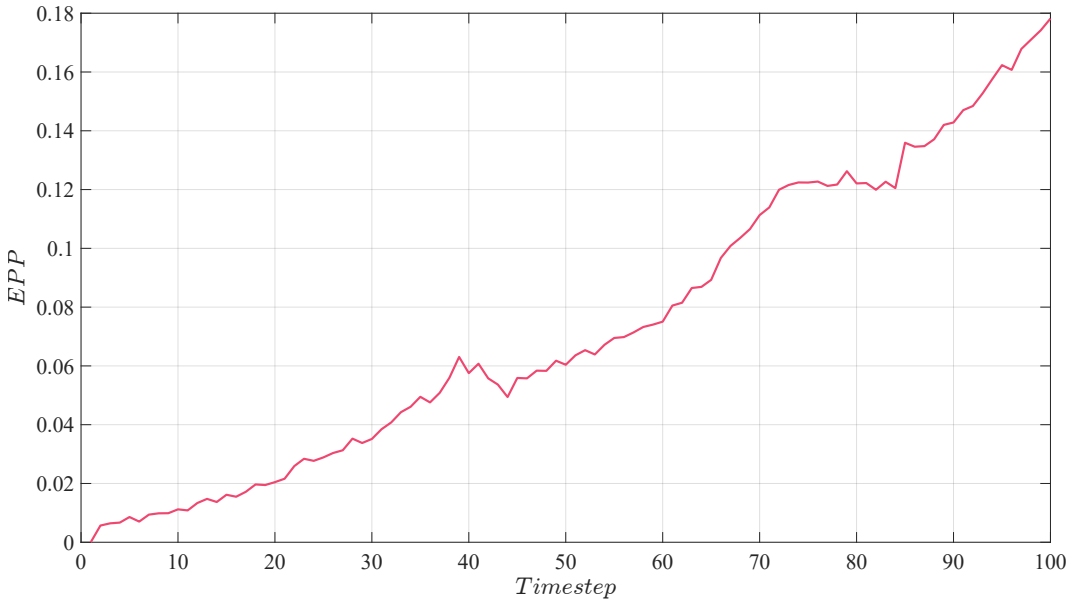


Figure 24: Errors corresponding to fitting an experimental worm

Of course, as the only available information from an experimental worm are its positions, the only computable error is the one corresponding to positions. The results are very promising, though not perfect: the general movement of the worm is properly captured and the error plot shows that the distance between experimental and predicted worms are not very high. However, at certain points, when the moments exerted caused a high curvature, the predicted worm started separating from the original one, as it was not able to get to those curvatures or to deduce the moments that caused them properly. This can be observed on figure 25, that plots five consecutive steps where the curvature in the experimental worm was not properly captured. This kind of differences, along with the ones shown in the synthetic tests, indicate the need to study the real values of $\boldsymbol{\eta}$ and to improve the model's accuracy.

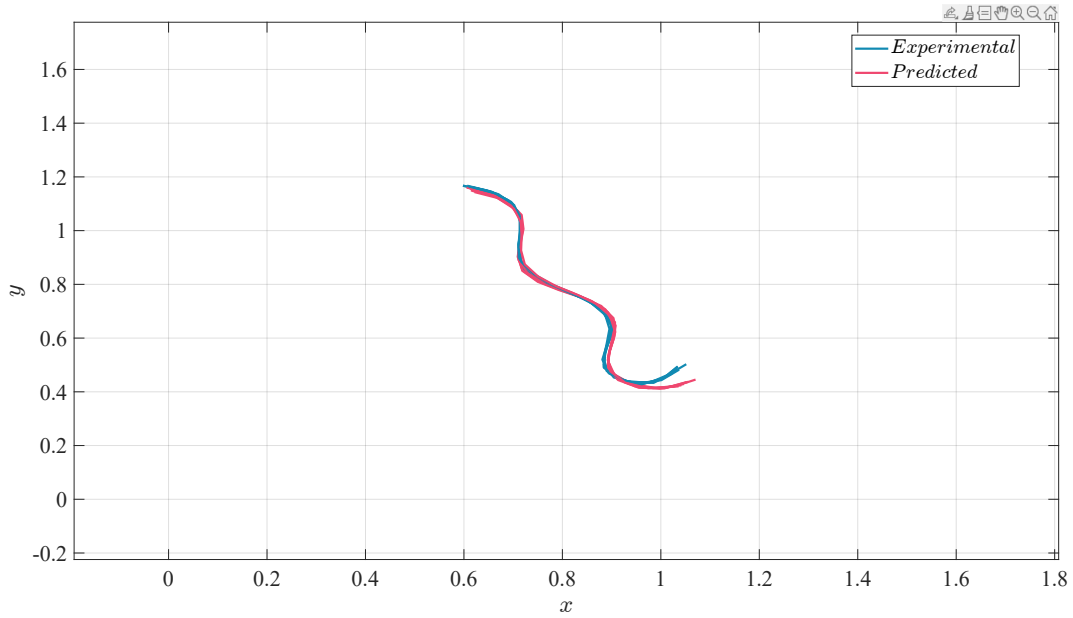


Figure 25: Fitting of the experimental worm on steps 10 to 15

6 Future Work

Even though a definitive answer to the question that wanted to be answered has not been found, what is true is that some light has been brought to the table on how should the problem be tackled and on its particularities. With this information, one has some hints on the direction that future investigations should take.

Due to the fundamental role of the values of the parameters of $\boldsymbol{\eta} = [\eta_t, \eta_n]$, it would be important either to have an estimation of those values using exterior means to the model, as computing the velocity of the center of masses and the proportion between the two components v_x and v_y, or to embed them as parameters for the model to deduce. An option to do so might be to create a staggered model that predicted the moments and then tried to find the best values of $\boldsymbol{\eta}$ given the found moments or inversely.

Another very useful idea is to use experimental positions as the initial positions to deform when creating the training, validation and test datasets. This would allow to train the models with real positions, that is, starting from a set of deformations that *C.elegans* can perform and at least one of them has once performed. This does not necessarily mean to use these initial positions as the only ones to take into account, as combining them with randomly generated initial positions could help in having a more general dataset.

In this regard, trying to find new ways of creating random moments or, at least, creating big datasets containing deformations caused by different kinds of moments could be useful to have a more representative training set, which would directly affect the model's capability to generalize and make better predictions.

The last step that could be taken would be to create the mathematical model, and its numerical implementation, of considering the worm to have dry friction. Then, some Neural Networks could be trained using data created with this new model. A comparison with the current hypothesis could be done, then.

7 Conclusions

What this project has shown is that, despite being far from a magical tool that can solve all your problems, machine learning has potential on problems of the one discussed here. The capacity of Neural Networks to capture and model the data in which they have been trained on, has been shown to be very effective. However, its limitations and the particular difficulties presented by the problem at hand have become apparent.

One of the most important notes to be taken is that the values of $\boldsymbol{\eta}$ are fundamental to the movement of the worm, as the dynamics that can appear are very different and that should be taken into account when creating the Datasets and designing the Machine Learning models. Without this knowledge or some sort of estimation of the value of the parameters, trying to fit properly the experimental data is similar to looking for a needle in a haystack.

On the other hand, the analysis performed indicate that, among the options tried, the best input was the one corresponding to NN2. That is, considering both the previous position $\boldsymbol{u_{n-1}}$ and the displacement $\boldsymbol{\Delta u}$. On regards to the method for creating the moments, the option that performed the best was the sum of sinusoidal waves, which at the end got expanded by also adding noise and considering constant and partial moments.

Another element that has detected to be quite problematic is the sensibility of the mathematical model near the limits of the admissible domain: small errors in the moments could lead to quite different results or, even, to not convergence.

What is also very important and validates the idea of using machine learning algorithms for this problem are the results shown when trying to fit an experimental worm. Of course, the results are far from perfect, but indicate that the approach can be successful if the steps mentioned in section 6 are followed.

All in all, whilst the results obtained are promising and indicate that Machine Learning algorithms could help in solving the treated issue, some limitations of the model and specificities of the problem at hand have raised. There is still much work to be done and much room for improvement if a conclusion and a definitive answer to the problem faced wants to be found.

References

[D22] Muñoz J.J. Condamin L. Doste D. On the net displacement of contact surface centroid in contractile bodies. *Mechanics Research Communications*, pages 1–5, 2022.

[Meh21] Bernhard Mehlig. *Machine Learning with Neural Networks*. Cambridge University Press, oct 2021.

[Mit20] Melanie Mitchell. *Artificial intelligence: a guide for thinking humans*. Picador, New York, 2020.

[ON15] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[Poy20] David Doste Poy. Numerical simulation of worm like motion. *TFM FME*, 2020.

[Sch19] Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. 11 2019.