The goal of this work is to explore the possibility of using Machine Learning and Artificial intelligence algorithms for GPS ephemeris compression.
This report is organized as follows. In part I the datased is introduced; in part II popular Machine Learning/AI methods for regression are outlined and an in-depth explanation of Feed Forward Neural Networks is delivered, followed in part III by their application for the ephemeris compression case. Finally, the results obtained are summarized and discussed in part IV.

Part I
The Data

The data used as showcase in this report are 14 days of the x-coordinate of the space vehicle 01 in the ECEF coordinate system from the 2019/09/10-00:00:00.000 obtained via both the official igs files (points are provided every $15min$) and a numerically propagated orbit (file *G01_orbit_ECEF_b.txt*) starting from the same initial conditions (provided every minute).
A plot of a smaller time window of x-coordinate, together with y and z, is shown in Fig.1 as function of time.

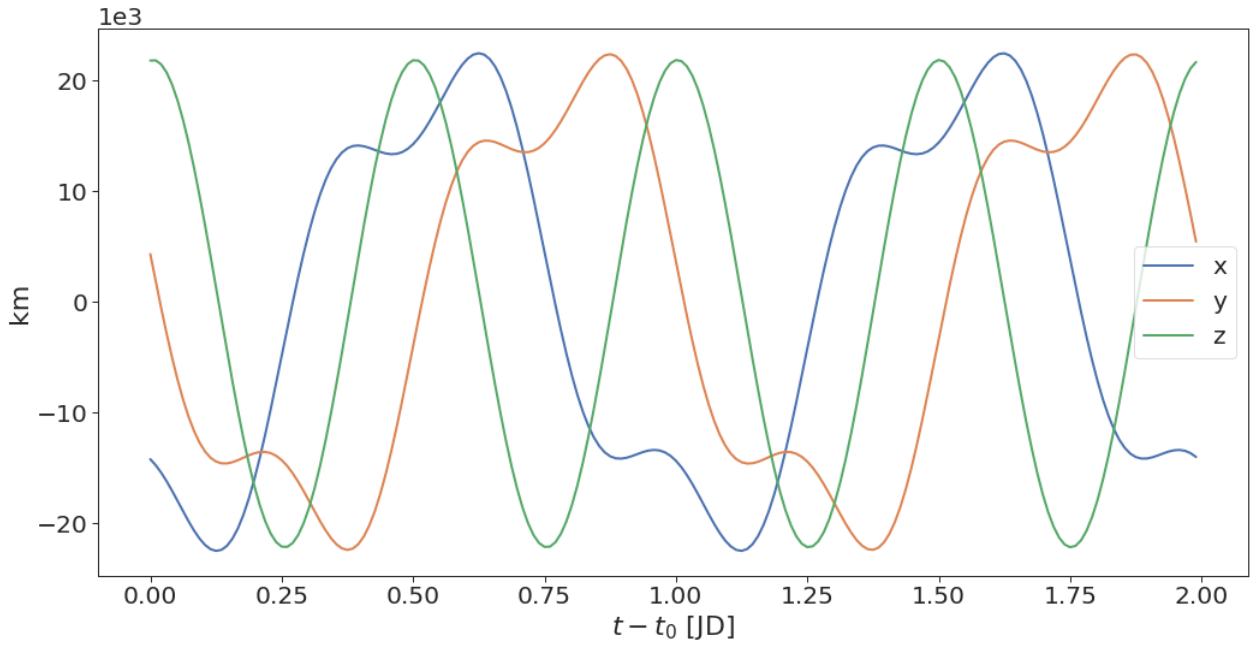

Figure 1: 2 days of propagated coordinates of the SV01 as function of time.

Propagation Error

The propagated orbits come with an error with respect to the actual igs orbits.
To compute it, first the propagated datased has to be decimated in order to obtain the datapoints corresponding to the times provided by the igs file (time steps of 15 minutes).
Then the difference in the x-coordinate and the radius as function of time t

$$e(t) = |x_{prop}(t) - x_{igs}(t)|, \qquad e(t_0) = 0m \tag{1}$$

(where the subscript $prop$ indicates data from the propagated file and t_0 the initial time) can be computed for every point and is shown as function of the propagation time $t - t_0$ in Fig. 2.

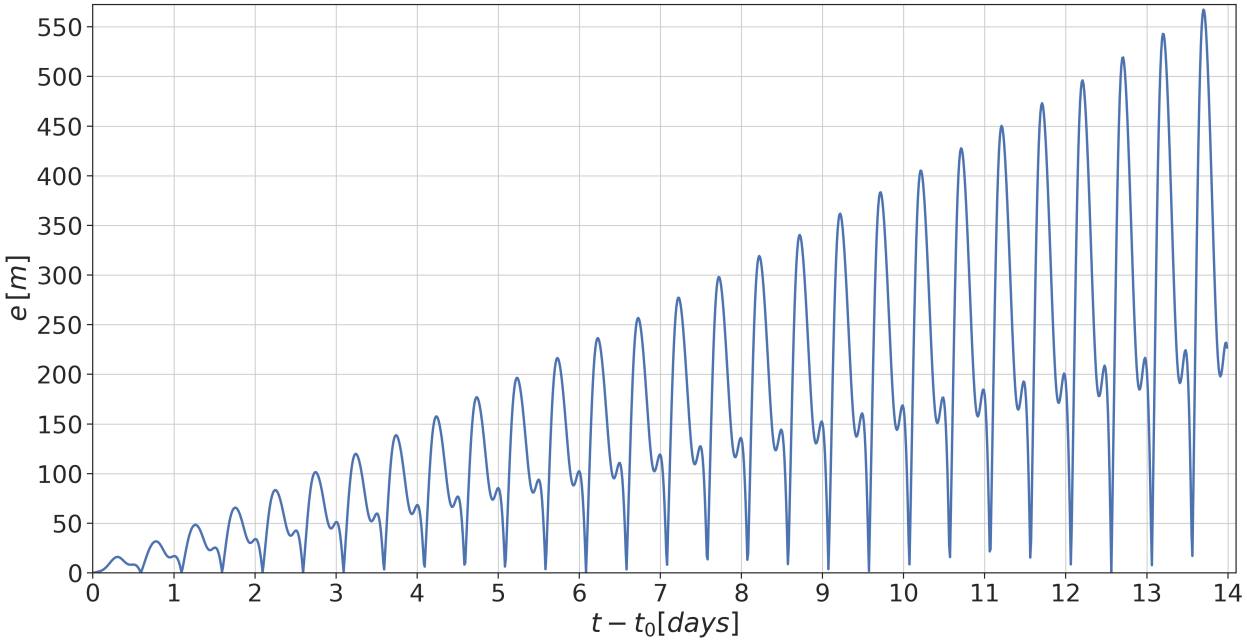

Figure 2: 14 days propagation error wrt the igs ephemeris (Eq. 1).

$t_f - t_0$	**Mean Error [m]**	**Maximum Error [m]**	$e(t_f - t_0)$ **[m]**
1 Day	13.205	31.773	16.651
2 Days	23.349	65.685	33.354
5 Days	54.978	176.855	84.644
7 Days	76.749	256.725	119.032
10 Days	110.215	383.299	168.726
14 Days	156.205	567.3	226.818
21 Days	241.843	941.201	283.849
28 Days	333.866	1369.13	228.611

Table 1: . Statistical Properties of the propagation errror (Eq. 1), for different propagation time duration. All errors are in meters, t_0 indicates the initial time instant, and t_f the final one. The propagation time duration is then $t_f - t_0$.

The goal of this work is to elaborate a compression scheme of such **propagated** data, eventually considering the error of propagation e.
Possible solutions comprehend different machine learning/AI algorithms.

Part II
Machine Learning & Neural Networks

Machine learning is the automated detection of patterns in data.
This field is branched into several subfields dealing with different types of learning tasks.

Given the nature of the orbit data, the focus in this work is on *Supervised Learning*, whose goal is to extrapolate information from some training data (data used to train the network, i.e. make it learn the patterns in the data) and use it to to reconstruct it or making prediction on unknown data (the simplest example of this branch of learning is linear regression).

Since the orbits consist of spatial coordinates given at different time instants, the problem of compression is recasted as a regression problem: the question is if it is possible to represent the coordinates using a function of time defined by a smaller number of coefficients than the original number of data points.

As one may expect, Machine Learning and Artificial Intelligence comprehend many different algorithms to takle this kind of task, so before starting with the most promising architectures, here is a small list of some very popular algorithms for regression which cannot be used for the presented task:
Decision-Based Algorithms (e.g. RandomForest [1], DecisionTree [8]), Gradiend Boosting (e.g. XGBoost [2]) and Gaussian Process [15] are able to provide extremely accurate predictions, but require a large amount of memory to be stored (easily more than the original data).

Now that these are out of the way, here are presented what are considered the best models for this work: Polynomial Regression and Feedforward Neural Networks.

II.1 Polynomial Regression

The simplest Machine Learning approach to regression is to model the relationship between an independent variable x and the dependent variable y as a polynomial or order n in x:

$$y \approx \sum_{j=0}^{n} c_j x^j \tag{2}$$

The vector of estimated polynomial regression coefficients $\boldsymbol{c} \equiv (c_0, c_1, c_2, ..., c_n)^T$ is obtained by minimizing the sum of squared residuals between the true y values and the approximating polynomial:

$$\boldsymbol{c} = argmin_{\boldsymbol{c}} \sum_{i=1}^{m} \left(\sum_{j=0}^{n} c_j x_i^j - y_i \right)^2 \tag{3}$$

where n indicates the degree of the polynomial and m the number of data points.
Since the scope of this work is the approximation of some given data, it is required that $n + 1 < m$ (the $n + 1 = m$ case is usually refered to as interpolation).

This is the Least Squares Regression.

Computing the gradient of the argument of the $argmin$ function in Eq.3 wrt c_k and setting it to zero it is found

$$2 \sum_{i=1}^{m} \left[\left(\sum_{j=0}^{n} c_j x_i^j - y_i \right) x_i^k \right] = 0 \tag{4}$$

Defining $\boldsymbol{P}(x_i) \equiv (1, x_i, x_i^2, ..., x_i^n)^T$, $\mathbf{0} = (0, 0, ..., 0)^T$ and indicating with $\langle \cdot, \cdot \rangle$ the scalar product, Eq. 4

can be rewritten $\forall k$ as

$$\mathbf{0} = \sum_{i=1}^{m} \Big(\langle \boldsymbol{c}, \boldsymbol{P}(x_i) \rangle - y_i \Big) \boldsymbol{P}(x_i) \tag{5}$$

Setting

$$\hat{A} \equiv \Big(\sum_{i=1}^{m} \boldsymbol{P}(x_i) \boldsymbol{P}^T(x_i) \Big), \qquad \hat{A}_{jk} = \sum_{i=1}^{m} P_j(x_i) P_k(x_i) \tag{6}$$

$$\boldsymbol{b} \equiv \sum_{i=1}^{m} y_i \boldsymbol{P}(x_i) \tag{7}$$

($\hat{A}$ is a symmetric $(n+1) \times (n+1)$ matrix) the solution is

$$\boldsymbol{c} = \hat{A}^{-1} \boldsymbol{b} \tag{8}$$

The main advantage of this approach with respect to other artificial intelligence (see below) is its simplicity: it is a convex optimization problem and requires only one parameter, the degree of the approximating polynomial.

Weighted Least Squares Polynomial Regression

If at every data point is associated a weight w_i, Eqs. 3-13 become:

$$\boldsymbol{c} = argmin_{\boldsymbol{c}} \sum_{i=1}^{m} w_i \Big(\sum_{j=0}^{n} c_j x_i^j - y_i \Big)^2 \tag{9}$$

$$2 \sum_{i=1}^{m} \Big[w_i \Big(\sum_{j=0}^{n} c_j x_i^j - y_i \Big) x_i^k \Big] = 0 \tag{10}$$

$$\hat{A} \equiv \Big(\sum_{i=1}^{m} w_i \boldsymbol{P}(x_i) \boldsymbol{P}^T(x_i) \Big), \qquad \hat{A}_{jk} = \sum_{i=1}^{m} w_i P_j(x_i) P_k(x_i) \tag{11}$$

$$\boldsymbol{b} \equiv \sum_{i=1}^{m} w_i y_i \boldsymbol{P}(x_i) \tag{12}$$

$$\boldsymbol{c} = \hat{A}^{-1} \boldsymbol{b} \tag{13}$$

II.2 Feedforward Neural Networks

An artificial neural network is a model of computation inspired by the structure of neural networks in the brain.
It consists of computing devices (neurons) connected to each other in a complex communication network, through which the brain is able to carry out highly complex computations.

According to the scope, there are several kind of artificial neural network, the most popular ones being:

- FeedForward Neural Networks
- Recurrent Neural Networks
- Convolutional Neural Networks

Where Recurrent and Convolutional Neural Networks are not considered in this work, as they are heavy, specialized models for tasks other than regression, i.e. sequence modelling and pattern recognition [13], [7].

Thus, only Feedforward Neural Networks are treated.

Structure & Notation

A Feedforward Neural Network (FFNN) can be described as an acyclic directed graph whose nodes v correspond to neurons and edges correspond to links between them. Each neuron, modeled as a simple scalar function, $\sigma : \mathbb{R} \to \mathbb{R}$ (refered to as *activation function*), receives as input a weighted sum of the outputs of the neurons connected to its incoming edges.

Such an ensamble is organized in layers, i.e. the set of neurons can be decomposed into a union of (non empty) disjoint subsets $V_t : t = 0, ...T$. In this way, the input of each neuron in layer V_t depends only on the output of neurons in the previous layer V_{t-1}.

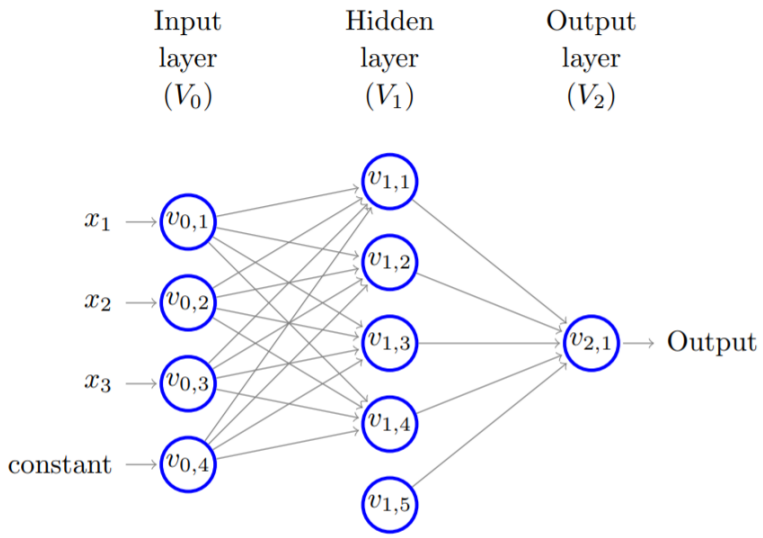

Figure 3: Feedforward Neural Network with 1 Hidden Layer.

With $v_{t,i}$ is denoted the *i*-th neuron of the *t*-th layer and with $O_{t,i}(\boldsymbol{x})$ the output of $v_{t,i}$ when the network is fed with the input vector $\boldsymbol{x} = (x_1, x_2, ...x_d)^T$.
Therefore, for the input layer V_0, for $i \in \{1, ..., d\}$, $O_{0,i}(\boldsymbol{x}) = x_i$ and for $i = d + 1$, $O_{0,i}(\boldsymbol{x}) = 1$.
Layers $V_1, ..., V_{T-1}$ are called hidden layers. The top layer, V_T , is called the output layer. In simple prediction problems (whether classification or regression) the output layer contains a single neuron whose output is the output of the network. We refer to T as the number of layers in the network (excluding V_0), or the "depth" of the network.
The size of the network is $|V|$, i.e. the total number of neurons. The "width" of the network is $\max_t |V_t|$.
An illustration of a layered feedforward neural network of depth 2, size 10, and width 5, is given in Fig. 3. Note there is a neuron in the hidden layer ($v_{1,5}$) that has no incoming edges: this will output the constant $\sigma(0)$.

The notation used to describe neural networks is summarized in Table 2.

Variable	Definition
$V_t, \quad t = 0, ..., T$	t-th layer
$d^{(t)} + 1$	number of neurons in layer t ($+1$ is to account for the constant neuron)
$v_{t,i}$	i-th neuron in the t-th layer
$O_{t,j}(\boldsymbol{x})$	output of neuron $v_{t,j}$ when $\boldsymbol{x}$ is fed to the network
$\boldsymbol{v}^{(t)} = \left(1, v_{t,1}, ..., v_{t,d^{(t)}}\right)^T$	vector of all neurons of layer t
$\boldsymbol{O}^{(t)}(\boldsymbol{x}) = \left(O_{t,1}(\boldsymbol{x}), ..., O_{t,d^{(t+1)}}(\boldsymbol{x})\right)^T$	vector of outputs of all the neurons of layer t when $\boldsymbol{x}$ is fed to the network
$w_{rj}^{(t+1)}$	weights of arc from neuron r of layer t to neuron j of layer $t+1$
$\boldsymbol{w}_j^{(t)} = \left(w_{0,j}^{(t)}, w_{1,j}^{(t)}, ..., w_{d^{(t)},j}^{(t)}\right)^T$	vector of all weights of arcs in input to neuron j of layer t
$W^{(t)} = \begin{pmatrix} w_{0,1}^{(t)} & w_{0,2}^{(t)} & \cdots & w_{0,d^{(t)}}^{(t)} \\ w_{1,1}^{(t)} & w_{1,2}^{(t)} & \cdots & w_{1,d^{(t)}}^{(t)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d^{(t-1)},1}^{(t)} & w_{d^{(t-1)},2}^{(t)} & \cdots & w_{d^{(t-1)},d^{(t)}}^{(t)} \end{pmatrix}^T$	matrix of weights of all arcs incoming to layer t

Table 2: Feedforward Neural Network notation

Forward Propagation

The output of the network is the output of the neurons in the output layer (for the Network in Fig. 3 it is the output of the neuron $v_{2,1}$).
This obviously depends on the outputs of the neurons in the previous layer, and the same is true for V_{T-1} and so on, i.e. the input of the network is forward propragated through the different layers.
So, the calculation of the output is achieved in a layer by layer manner, starting from the input one.

Suppose the outputs of the neurons at layer t when a vector $\boldsymbol{x}$ is given in input to the network has already been computed.
Then, it is possible calculate the outputs of the neurons at layer $t+1$.
Fix some $v_{t+1,j} \in V_{t+1}$.
Let $a_{t+1,j}(x)$ denote the input to $v_{t+1,j}$ when the network is fed with the input vector $\boldsymbol{x}$.
Then ($\langle \cdot, \cdot \rangle$ indicates the scalar product),

$$a_{t+1,j}(\boldsymbol{x}) = < \boldsymbol{w}_j^{(t+1)}, \boldsymbol{O}^{(t)}(\boldsymbol{x}) > \tag{14}$$

$$O_{t+1,j}(\boldsymbol{x}) = \sigma\left(a_{t+1,j}(\boldsymbol{x})\right) \tag{15}$$

Or, considering all neurons in layer $t+1$

$$\boldsymbol{O}^{(t+1)}(\boldsymbol{x}) = \sigma\left(W^{(t+1)} \cdot \boldsymbol{O}^{(t)}(\boldsymbol{x})\right) \tag{16}$$

That is, the input to $v_{t+1,j}$ is a weighted sum of the outputs of the neurons in V_t that are connected to $v_{t+1,j}$, where weighting is according to w, and the output of $v_{t+1,j}$ is simply the application of the

activation function σ on its input.

Thus, to compute the output of the network, this calculation is repeated for every neuron in every layer, starting from the V_1 (the output of V_0 is just the $\boldsymbol{x}$, plus a constant term):

Algorithm 1 Forward Propagation

function FORWARD PROPAGATION($\boldsymbol{x}$)
  x: d-dimensional input sample

  Compute the output of V_0:

$$\boldsymbol{O}^{(0)}(\boldsymbol{x}) = (1, x_1, ..., x_d)^T$$

  for $t = 1, 2, ..., T$ **do**
    Compute output of layer t:

$$\boldsymbol{O}^{(t)}(\boldsymbol{x}) = \sigma\Big(W^{(t)} \cdot \boldsymbol{O}^{(t-1)}(\boldsymbol{x})\Big)$$

  Compute the output of the network

$$\boldsymbol{y} = \boldsymbol{O}^{(T)}(\boldsymbol{x})$$

  return $\boldsymbol{y}$

To make things more clear, and explicitly show that the outputs of neural networks are essentially non linear combinations of the neurons activation funcions of the input, consider the toy example with: a scalar input x, a scalar output y and a network composed by 2 hidden layers with 3 neurons with sine activation each and no constant neurons (for simplicity, the same argument can be easily extendend to the case with constant neurons).
Then, the output of the first hidden layer V_1 is:

$$\begin{cases} O_{1,1}(x) = \sin\left(w_{1,1}^{(1)} x\right) \\ O_{1,2}(x) = \sin\left(w_{1,2}^{(1)} x\right) \\ O_{1,3}(x) = \sin\left(w_{1,3}^{(1)} x\right) \end{cases} \tag{17}$$

For the second hidden layer:

$$\begin{cases} O_{2,1}(x) = \sin\left(w_{1,1}^{(2)} O_{1,1}(x) + w_{2,1}^{(2)} O_{1,2}(x) + w_{3,1}^{(2)} O_{1,3}(x)\right) \\ O_{2,2}(x) = \sin\left(w_{1,2}^{(2)} O_{1,1}(x) + w_{2,2}^{(2)} O_{1,2}(x) + w_{3,2}^{(2)} O_{1,3}(x)\right) \\ O_{2,3}(x) = \sin\left(w_{1,3}^{(2)} O_{1,1}(x) + w_{2,3}^{(2)} O_{1,2}(x) + w_{3,3}^{(2)} O_{1,3}(x)\right) \end{cases} \tag{18}$$

Then, setting a linear activation function $\sigma(a) = a$ in the output layer neuron, the output of the network is:

$$y = w_{1,1}^{(3)} O_{2,1}(x) + w_{2,1}^{(2)} O_{2,2}(x) + w_{3,1}^{(2)} O_{2,3}(x) \tag{19}$$

From such discussion, it is clear that the structure and the functioning of a Feedforward Neural Network depends on different parameters: the activation function σ, the number of layers T and the number of

neurons in every layer. These will be treated later.

Now the question is how learning is achieved.

Learning Process - Backpropagation

In prediction problems, regression in this case, the goal is, given some data $(\boldsymbol{x_1}, y_1), (\boldsymbol{x_2}, y_2), .., (\boldsymbol{x_m}, y_m)$, to find the weights w that minimize a loss function

$$L(w) = \frac{1}{m} \sum_{i=1}^{m} \ell(w, (\boldsymbol{x_i}, y_i)) \tag{20}$$

If at every data point is associated an importance I_i (the equivalent of the weights in weighted polynomial regression), the loss becomes

$$L(w) = \frac{1}{\sum_{i=1}^{m} I_i} \sum_{i=1}^{m} I_i \ell(w, (\boldsymbol{x_i}, y_i))$$

In the following treatment, for ease of notation, the loss is given by Eq. 20.

One may think, as an analogy, to a linear regression, where the goal is finding the best parameters a, b such that the mean squared difference between y and $ax + b$ is minimum (note when talking about neural networks the term *weights* is used in place of *coefficients*).
The key difference, however, is that in the case of neural networks the loss is a non-convex function of the weights, with possibly many local minima. This means there isn't an analytic solution to the minimization of the loss function.
This has to be achieved via an iterative procedure, the simplest one being *Stochastic Gradient Descent*. The idea is:

1. Forward propagate every data point $\boldsymbol{x_i}$, $i = 1, .., m$ and compute the loss L with respect to y_i;
2. Compute the gradient of the loss with respect to the weights;
3. Update the weights moving in the opposite direction of such gradient;
4. Repeat from 1. from a number of iterations (to be set in order to reach convergence)

The most difficult part is step 2, i.e the computation of the gradient, (the term backpropagation strictly refers only to the algorithm for computing the gradient) as the derivative of the loss with respect to each single weight in the network is required, but it is possible to compute the loss only after the last layer.

The update rule for the weight connecting neuron i in layer $t-1$ to neuron j in layer t at time step s reads as (s indicates the iteration number):

$$w_{ij}^{(t)(s+1)} = w_{ij}^{(t)(s)} - \eta \frac{\partial L_s}{\partial w_{ij}^{(t)(s)}} \tag{21}$$

$$\frac{\partial L_s}{\partial w} = \frac{1}{m} \left(\sum_{i=1}^{m} \frac{\partial \ell(w, (\boldsymbol{x_i}, y_i))}{\partial w} \right) \tag{22}$$

where η is the step size, also called *learning rate*. The gradient is decomposed using the chain rule (**Note**: to compact the notation here $v_{t,i}$ represents also the output of the neuron i of layer t):

$$\frac{\partial L_s}{\partial w_{ij}^{(t)}} = \frac{\partial L}{\partial a_{t,j}} \frac{\partial a_{t,j}}{\partial w_{ij}^{(t)}} = \delta_j^{(t)} \frac{\partial}{\partial w_{ij}^{(t)}} \left(\sum_{k=0}^{d^{(t-1)}} w_{kj}^{(t)} v_{t-1,k} \right) = \delta_j^{(t)} v_{t-1,i} \tag{23}$$

$$\delta_j^{(t)} = \frac{\partial L}{\partial v_{t,j}} \frac{\partial v_{t,j}}{\partial a_{t,j}} = \frac{\partial L}{\partial v_{t,j}} \frac{\partial \sigma}{\partial a_{t,j}} \tag{24}$$

Now it is required to compute the variation of the loss w.r.t. $v_{t,j}$, remembering that:

- a change in layer t affects only neurons in layer $t+1$, and then each following layer up to the output;
- each neuron can affect all the neurons in the next layer;
- to compute $\delta^{(t)}$, $\delta^{(t+1)}$ is needed

$$\frac{\partial L}{\partial v_{t,j}} = \sum_{k=1}^{d^{(t+1)}} \frac{\partial L}{\partial a_{t+1,k}} \frac{\partial a_{t+1,k}}{\partial v_{t,j}} = \sum_{k=1}^{d^{(t+1)}} w_{j,k}^{(t+1)} \delta_k^{(t+1)} \tag{25}$$

$$\implies \delta_j^{(t)} = \frac{\partial \sigma}{\partial a_{t,j}} \sum_{k=1}^{d^{(t+1)}} w_{j,k}^{(t+1)} \delta_k^{(t+1)} \tag{26}$$

The solution for every layer requires the solution of the following one, so the strategy is starting from the last layer ($\delta^{(T)}$ can be computed from the loss on the output) and then *backpropagate* the gradients through all the layers down to the first.

Algorithm 2 Backpropagation

function BP(data point,$\{w_{ij}\}$)
    data point: $(\boldsymbol{x_k}, y_k)$
    $\{w_{ij}\}$: all weights of the neural network

    compute $a^{(t)}$ and $v^{(t)}$ for $t = 1, .., T$ (forward propagation)
    compute $\delta^{(T)} = \frac{\partial L}{\partial a^{(T)}}$
    for $t = T-1, T-2, ..., 1$ **do**

$$\delta_j^{(t)} = \frac{\partial \sigma}{\partial a_{t,j}} \sum_{k=1}^{d^{(t+1)}} w_{j,k}^{(t+1)} \delta_k^{(t+1)} \quad \forall j = 1, ..., d^{(t)}$$

    return $\delta_j^{(t)} \quad \forall j, t$

Thus, the complete neural network learning algorithm is

Algorithm 3 Learning Neural Networks with Stochastic Gradient Descent

function BP-SGD(data points,η,τ)
    data points: $(\boldsymbol{x_1}, y_1), (\boldsymbol{x_2}, y_2), .., (\boldsymbol{x_m}, y_m)$
    η: learning rate
    τ: number of iterations

    Initialize $w_{ij}^{(t)} \quad \forall i, j, t$
    for $s = 0, 1, 2, ..., \tau$ **do**
        pick randomly a sample (x_k, y_k)
        compute the output of every neuron $v_{t,j}$using forward propagation
        compute $\delta_j^{(t)} \quad \forall j, t$ backpropagating the gradient
        update the weights

$$w_{ij}^{(t)(s+1)} = w_{ij}^{(t)(s)} - \eta v_{t-1,i} \delta_j^{(t)} \quad \forall i, j, t$$

    return $w_{ij}^{(t)} \quad \forall i, j, t$

Neural Network Parameters

From the discussion above, it is easy to see that a neural network requires many parameters, both for defining its structure and the learning process, that heavily impact the approximation performance.

Structural parameters

Number of hidden layers: the input and output layers are necessary and fixed, but number of hidden ones are arbitrary. Usually for *deep* neural network it is meant a network with more than 2 hidden layers;

Number of neurons: The number of units in every hidden layer must be tuned, while their number in the input and output layers are set by the data;

Activation Function $\sigma(a)$: Various activation functions can be exploited; the most popular ones are

Sign function $$\sigma(a) = \begin{cases} -1 & a < 0 \\ 0 & a = 0 \\ 1 & a > 0 \end{cases}$$

Threshold function $$\sigma(a) = \begin{cases} 0 & a < 0 \\ 1 & a \geqslant 0 \end{cases}$$

Sigmoid function $$\sigma(a) = \frac{1}{1 + exp(-a)}$$

Hyperbolic Tangent $$\sigma(a) = \tanh a$$

Rectified Linear Unit (ReLU) $$\sigma(a) = \max(0, a)$$

Linear function $$\sigma(a) = a$$

Learning parameters

Learning rate η: it controls how much to change the model in response to the loss each time the model weights are updated.

Optimization Algorithm: in this report only the simplest optimization algorithm is explained, Stochastic Gradient Descent, but in practice more advanced ones are implemented, like Adam or RMSProp; for more refer to [11].

Number of Learning Iterations

All these parameters have to be fixed before the learning process starts.
Unfortunatelly, there is not a rule for this choice, so they must be chosen empirically, usually using a grid search, which consists in an iteration through different combinations of them, performing the learning procedure described above on each one and then choosing as best the one which provides the best approximation performances.

II.3 Extreme Learning Machines

Extreme Learning Machines (ELM) [5] are Feedforward Neural Networks, trained not via gradient-based backpropagation, but computing (only once) the Moore-Penrose generalized inverse of a matrix to set its weights.
Thus, the learning procedure is *extremely fast* (hence the adjective "Extreme" in ELM).

Structure & Forward Propagation

An ELM is a Single-layer Feedforward Neural Network (SLFN) (See Fig. 3).
The key difference with respect to their "traditional" counterpart is that the weights going from the input to the hidden layer ($\boldsymbol{W}^{(1)}$) don't need to be tuned, they can be chosen arbitrarily (even randomly).

Before going into details, here it is explained how the output of an ELM with N neurons in the hidden layer is computed when a data point x_i is fed to it.
Using the same notation of Table 2, the output of the hidden layer V_1 is:

$$\boldsymbol{O}^{(1)}(x_i) = \sigma(\boldsymbol{W}^{(1)} x_i + \boldsymbol{b}^{(1)}) \tag{27}$$

where, since the input layer is made of just one neuron, $\boldsymbol{W}^{(\mathbf{1})} = (w_{0,1}^{(1)} w_{0,2}^{(1)} \cdots w_{0,N}^{(1)})$ (reminding it indicates all the weights of arcs connecting the input layer neurons to the hidden layer ones) is just a vector and $\boldsymbol{b}^{(1)} = (b_1, b_2, \cdots b_N)^T$ indicate the *bias*, i.e a constant vector.
Then, the output of the network is (the output layer activation function is linear $\sigma(a) = a$):

$$O^{(2)}(x_i) \equiv \tilde{y}_i = \boldsymbol{W}^{(2)} \boldsymbol{O}^{(1)}(x_i) \tag{28}$$

In this case, since the output is scalar, $\boldsymbol{W}^{(2)} = (w_{1,0}^{(2)} w_{2,0}^{(2)} \cdots w_{N,0}^{(2)})$ (which indicates all the weights of arcs connecting all the neurons in the hidden layer and the output neuron), is just a (horizontal) vector.

It is possible to rewrite the equation in matrix form, considering m data points $(x_i, y_i),\ i = 1, ...m$ and defining

$$\boldsymbol{X} = (x_1, x_2, ..., x_m)^T, \qquad \boldsymbol{Y} = (y_1, y_2, ..., y_m)^T \tag{29}$$

$$\tilde{\boldsymbol{Y}} = (\tilde{y}_1, \tilde{y}_2, ..., \tilde{y}_m)^T \tag{30}$$

$$\implies \tilde{\boldsymbol{Y}}^T = \boldsymbol{W}^{(2)} H^T \tag{31}$$

Where

$$H = \begin{pmatrix} \sigma(w_{0,1}^{(1)} x_1 + b_1) & \cdots & \sigma(w_{0,N}^{(1)} x_1 + b_N) \\ \sigma(w_{0,1}^{(1)} x_2 + b_1) & \cdots & \sigma(w_{0,N}^{(1)} x_2 + b_N) \\ \vdots & \ddots & \vdots \\ \sigma(w_{0,1}^{(1)} x_m + b_1) & \cdots & \sigma(w_{0,N}^{(1)} x_m + b_N) \end{pmatrix} \tag{32}$$

is a $m \times N$ matrix.
So, the complete equation is:

$$\begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \vdots \\ \tilde{y}_m \end{pmatrix} = \begin{pmatrix} \sigma(w_{0,1}^{(1)} x_1 + b_1) & \cdots & \sigma(w_{0,N}^{(1)} x_1 + b_N) \\ \sigma(w_{0,1}^{(1)} x_2 + b_1) & \cdots & \sigma(w_{0,N}^{(1)} x_2 + b_N) \\ \vdots & \ddots & \vdots \\ \sigma(w_{0,1}^{(1)} x_m + b_1) & \cdots & \sigma(w_{0,N}^{(1)} x_m + b_N) \end{pmatrix} \begin{pmatrix} w_{1,0}^{(2)} \\ w_{2,0}^{(2)} \\ \vdots \\ w_{N,0}^{(2)} \end{pmatrix} \tag{33}$$

Learning ELMs

The key difference between ELMs and other SLFNs is that the former weights of arcs connecting the input layer to the hidden one are not to be learned: they are arbitrary; one can use a particular rule to set them or just randomly sampling them from some distribution.
Thus an ELM is a linear system in the weights to be learned (Eq.57), and this enables the use of the following theorem [5],[12] to compute them minimizing the mean squared error between the output of the network $\tilde{\boldsymbol{Y}}$ and the data target values $\boldsymbol{Y}$, $1/m||\tilde{\boldsymbol{Y}} - \boldsymbol{Y}||^2$:

> Let there exist a matrix G such that $G\boldsymbol{y}$ is a minimum 2-norm least-squares solution of a linear system $A\boldsymbol{x} = \boldsymbol{y}$. Then it is necessary and sufficient that $G = A^+$, the Moore-Penrose generalized inverse of matrix A.

Then, given some data points $(x_1, y_1), \cdots, (x_m, y_m)$, the least-squares solution:

$$\boldsymbol{\beta} = argmin_{\boldsymbol{W}^{(2)}} \left\| H\boldsymbol{W}^{(2)T} - \boldsymbol{Y} \right\|^2 \tag{34}$$

where $\boldsymbol{Y} = (y_1, \cdots y_m)^T$, is given by:

$$\boldsymbol{\beta} = H^+\boldsymbol{Y} \tag{35}$$

where H^+ is the $(N \times m)$ Moore-Penrose generalized inverse matrix of H, i.e. it satisfies

$$HH^+H = H,\ H^+HH^+ = H^+,\ (HH^+)^T = HH^+,\ (H^+H)^T = H^+H \tag{36}$$

Now it is easy to see the great advantage ELMs have with respect to other SLFNs:

1. The solution for the weights that minimize the mean squared error loss is obtained analitically;
2. There are few structural parameters to be tuned.

Reformulation of ELM Learning as Least Squares Regression

Here it is explicitly shown that ELMs are a just fancy reformulation of a classical least squares regression.

Let's start from the output of an ELM $\tilde{\boldsymbol{Y}} = (\tilde{y_1}, ..., \tilde{y_m})^T$ when $\boldsymbol{X} = (x_1, ..., x_m)^T$ is fed to the network (Eq.31):

$$\tilde{\boldsymbol{Y}} = H\boldsymbol{W}^{(2)T}, \tag{37}$$

$$\text{or} \qquad \tilde{y_i} = \langle \boldsymbol{H_i}, \boldsymbol{W}^{(2)T} \rangle \tag{38}$$

where H is given by Eq. 32, $\boldsymbol{H_i}$ indicates its *ith* row and $\langle \cdot, \cdot \rangle$ the scalar product.

Renaming for simplicity:

$$\boldsymbol{W}^{(2)T} \qquad \longrightarrow \qquad \boldsymbol{c} = (c_1, ..., c_N)^T \tag{39}$$

$$\tilde{y_i} = \sum_{j=1}^{N} c_j H_{ij} \tag{40}$$

where H_{ij} indicates the *ith* row jth column element of H.

The estimated weights $\boldsymbol{c}$ are obtained by minimizing the sum of squared residuals (or, equivalently, the mean squared error) between the true $\boldsymbol{y}$ values and the approximated ones $\tilde{\boldsymbol{y}}$:

$$\boldsymbol{c} = argmin_{\boldsymbol{c}} \sum_{i=1}^{m} \left(\langle \boldsymbol{H_i}, \boldsymbol{c} \rangle - y_i \right)^2 \tag{41}$$

where, as usual $N < m$.

Computing the derivative of the argument of the *argmin* with respect to c_k and setting it to zero it is found:

$$2\sum_{i=1}^{m}\Big(\langle \boldsymbol{c}, \boldsymbol{H_i}\rangle - y_i\Big)\boldsymbol{H_i}^T = \boldsymbol{0} \tag{42}$$

Defining:

$$\hat{A} \equiv \sum_{i=1}^{m} \boldsymbol{H_i}^T \boldsymbol{H_i}, \qquad \hat{A}_{jk} = \sum_{i=1}^{m} H_{ij} H_{ik} \tag{43}$$

$$\boldsymbol{b} \equiv \sum_{i=1}^{m} y_i \boldsymbol{H_i}^T \tag{44}$$

$$\implies \boldsymbol{c} = \hat{A}^{-1}\boldsymbol{b} \tag{45}$$

where $\hat{A}$ is a symmetric $(N \times N)$ matrix.

These correspond to the equations found for the polynomial least squares regression (Eqs. 4-13) with:

$$\boldsymbol{P}(x_i)^T = (x_i^0, x_i^1, x_i^2, ..., x_i^{N-1}) \qquad \longrightarrow \qquad \boldsymbol{H_i} \tag{46}$$

$$x_i^{j-1} \qquad \longrightarrow \qquad H_{ij} \tag{47}$$

Weighted ELM Regression

If performing a weighted ELM regression, where at every data point is associated a weight w_i, Eq. 41-45 become:

$$\boldsymbol{c} = argmin_{\boldsymbol{c}} \sum_{i=1}^{m} w_i \Big(\langle \boldsymbol{H_i}, \boldsymbol{c}\rangle - y_i\Big)^2 \tag{48}$$

$$2\sum_{i=1}^{m} w_i \Big(\langle \boldsymbol{c}, \boldsymbol{H_i}\rangle - y_i\Big)\boldsymbol{H_i}^T = \boldsymbol{0} \tag{49}$$

$$\hat{A} \equiv \sum_{i=1}^{m} w_i \boldsymbol{H_i}^T \boldsymbol{H_i}, \qquad \hat{A}_{jk} = \sum_{i=1}^{m} w_i H_{ij} H_{ik} \tag{50}$$

$$\boldsymbol{b} \equiv \sum_{i=1}^{m} w_i y_i \boldsymbol{H_i}^T \tag{51}$$

$$\boldsymbol{c} = \hat{A}^{-1}\boldsymbol{b} \tag{52}$$

where $\hat{A}$ is a symmetric $(N \times N)$ matrix.

Radial Basis Function ELMs

In the previous sections, it is assumed that the activations functions are additive, i.e.

$$\sigma(w_j, b_j, x_i) = \sigma(w_j x_i + b_j) \tag{53}$$

With $w_j, b_j \in \mathcal{R}$.
If, instead, the activation functions of the neurons in the hidden layer are Radial Basis Functions (RBFs), i.e.:

$$\sigma(w_j, b_j, x_i) = \sigma(b_j|x_i - w_j|) \tag{54}$$

the output of the network $\tilde{y}_i$ when x_i is fed to it just is a linear combination of these RBFs applied to the input x_i:

$$\tilde{y}_i = \sum_{j=1}^{N} c_j \sigma\Big(b_j|x_i - w_j|\Big) \tag{55}$$

$$= \sum_{j=1}^{N} c_j H_{ij} \tag{56}$$

Where

$$H = \begin{pmatrix} \sigma(b_1|x_1 - w_{0,1}^{(1)}|) & \cdots & \sigma(b_N|x_1 - w_{0,N}^{(1)}|) \\ \sigma(b_1|x_2 - w_{0,1}^{(1)}|) & \cdots & \sigma(b_N|x_2 - w_{0,N}^{(1)}|) \\ \vdots & \ddots & \vdots \\ \sigma(b_1|x_m - w_{0,1}^{(1)}|) & \cdots & \sigma(b_N|x_m - w_{0,N}^{(1)}|) \end{pmatrix} \tag{57}$$

The most popular choice for the activation is a Gaussian RBF:

$$\sigma(w_j, b_j, x) = \exp\Big(-b_j|x - w_j|^2\Big) \tag{58}$$

The Learning Paradigm (Eq. 35, 45 or 52) remains unchanged.

ELM Parameters

When building an ELM, the training procedure is fixed, so there are only some structural parameters to be tuned:

Number of neurons in the hidden layer N;

Activation function σ: in Eq. 32 all the neurons in the hidden layer have the same σ, but it is also possible to chose a different one for each neuron;

Input layer weights and biases: in the simplest ELM they are randomly sampled from a normal distribution. However, to improve performance they can be chosen deterministically according to the specific task.

II.4 Neural Networks and Polynomial Regression

In this section is presented a simple argument [3] that Neural Networks are essentially polynomial regression models with the effective polynomial degree growing at each hidden layer.

To demonstrate such correspondence, consider a toy example where the independent variable $\boldsymbol{x}$ is two-dimensional $\boldsymbol{x} = (x_1, x_2)^T$.
The inputs to neuron j in the first hidden layer, including the constant node, will take the form (mantaining the same notation as Table 2) $w_{0,j}^{(1)} + w_{1,j}^{(1)} x_1 + w_{2,j}^{(1)} x_2$ and $w_{3,j}^{(1)} + w_{4,j}^{(1)} x_1 + w_{5,j}^{(1)} x_2$.

Consider as activation function $\sigma(a) = a^2$.
Then outputs of that first layer will be quadratic functions of x_1 and x_2. The second layer will output fourth-degree polynomials, then degree eight, and so on.

Clearly, this works in a similar way for any polynomial activation function. So, for a polynomial activation function, minimizing the mean squared error in Neural Networks training corresponds to performing polynomial regression.

For general activation functions and implementations, it is possible to state at least that the function is close to a polynomial, as any continuous function on a compact set can be approximated uniformly by polynomials (Stone-Weierstrass Theorem [16]).
Another popular activation function is ReLU, $\sigma(a) = max(0, a)$ (or its variations), which is a piecewise polynomial.

In any case, for practical purposes, most activation functions can be approximated by a polynomial.

To summarize the argument, the following holds:

> If the activation function is a polynomial, or is implemented by one, a neural network performs a form of polynomial regression.
> The degree of the approximating polynomial increases from layer to layer.

Obviously, a similar argument holds even for Extremely Learning Machines, with the difference that in this case there is only one hidden layer.

This Neural Networks - Polynomial predictors correspondence suggests that in many applications, one might simply fit a polynomial model, instead of a neural network.
This would allow to avoid going thorugh all the complications that comes with Neural Networks learning, such as having to set many hyperparameters and deal with Backpropagation.

In [3] and [9] are presented a set of empirical comparisons of Neural Networks and polynomial regression models on different datasets; in all cases (apart from one), the performances of the polynomial regression matches, and often beats, that of neural networks.

Polynomial Regression from Neural Networks

It is also possible to obtain *exactly* a polynomial of order n regression with Feed Forward Neural Networks, by fixing the right architecture and activation funcions.

Precisely, this is obtained with a single hidden layer network (SLFN) with:

1. no biases (refering to Fig. 3, the neurons $v_{0,4}$ and $v_{1,5}$ are not present);
2. $n + 1$ hidden neurons;
3. activation function of the *k*th ($k = 0, ..., n$) hidden neuron set to $\sigma_k(x) = x^k$;
4. weights of arcs going from the input to the hidden layer $\boldsymbol{W}^{(1)}$ fixed to 1.

So, the output of a such a network when x is fed to it is (defining, as before,$\boldsymbol{P}(x) \equiv (1, x, x^2, ..., x^n)^T$):

$$\tilde{y} = \boldsymbol{W}^{(2)}\boldsymbol{P}(x) \tag{59}$$

$$= \sum_{k=0}^{n} w_{k,0}^{(2)} x^k \tag{60}$$

Renaming

$$w_{k,0}^{(2)} \qquad \longrightarrow \qquad c_k \tag{61}$$

One gets

$$\tilde{y} = \sum_{k=0}^{n} c_k x^k \tag{62}$$

which is exactly what is obtained by modelling $y(x)$ using a polynomial of order n (Eq. 2).

This argument can be generalized to any polynomial (e.g. Chebyshev Polynomials).

Since the weights going from the input layer to the hidden one are fixed, the weights $w_{k,0}^{(2)}$ (or, equivalently, the coefficients c_k) can be easily learned using the ELM algorithm (Eq. 35).

In this way, by definition, they will match exactly (a part from a negligible deviation due to the possibly different algorithms used to compute the inverse matrix) the ones obtained with polynomial Least Squares regression.

In general, the advantage of using an ELM or a FFNN is that they can take different kinds of activation functions in the same layer; for example one neuron might have a sine, and another the exponential, and so on.
This results in more flexibility.

Another difference is that the output of Multi-Layer Feedforward Neural Networks can be non-linear in terms of activation functions (see, for example Eq. 19). This, in theory, makes them capaple of learning more complex relationships in the data.

Part III
The Ephemeris Compression Case

The data used as showcase of what Feed Forward Neural Networks and Extreme Learning Machines can do in terms of ephemeris compression are 14 days of the propagated x-coordinate as function of time t (blue line in Fig. 1).
The coordinate is filtered to one point every 15 minutes, which correspond to $m = 96$ data points per day.
The goal is to find the network parameters that provide the lowest Root Mean Squared Error (RMS):

$$RMS = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(\tilde{x}_i - x_i)^2} \tag{63}$$

(where $\tilde{x}_i$ and x_i indicate the coordinate predicted by the network and the data ones, respectively), with the constraint that the total number of weights to be stored must be, in order to achieve compression, smaller than the number of data points (96).

Furthermore, before feeding it to the varius models, the time was rescaled to $[-1, 1]$.

This part is organized as follows: first an unweighted regression (without considering the error of propagation) is performed on one day of data, for different models (FeedForward Neural Networks, Extreme Learning Machines) and the results compared to a polynomial regression; then, the ones which provided the best performances are re-trained for multiple days of data.
Finally, on such models is performed a weighted regression.

III.1 Unweighted Regression

In this first stage, as the interest is just in finding a good configuration of network (FFNN and ELM) parameters and evaluate its performances with respect to a polynomial model, the propagation error e (Eq. 1) is not taken into account: all the points in the data have the same importance during model training.

1 Day of Data

One day of data correspond to $m = 96$ data points: (t_i, x_i), $i = 1, \cdots, 96$.
Here are found the best FFNN and ELM structural parameters, which will lead to the proposed method for ephemeris compression.
These results will also be used for more days of data and weighted regression.

Polynomial Regression

The target of comparison for Neural Networks is a polynomial model. The results obtained with polynomial least squaes regression (Eqs. 2-13) for several polynomial degrees N are reported in Table 3.

Polynomial Degree	Q	RMS [m]
26	27	2.801
28	29	0.613
30	31	0.129
32	33	0.031
34	35	0.021
36	37	0.02
38	39	0.02

Table 3: Polynomial Regression results. N indicates the polynomial degree and Q the number of coefficients to be stored, i.e. $1 + N$.

Feedforward Neural Networks

As stated above, to obtain good performances from a Neural Network, it is first required to find the best configuration of its structural and learning parameters.

Grid Search

The number of hiddel layers, the number of neurons N in each one of them and the activation functions are chosen performing a grid search[1].

The possible values of such parameters tried are:

Number of Hidden Layers	Number of Hidden Neurons	Activation Functions
1	10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20	sin, tanh, relu, sigmoid
2	[2,2], [3,3], [4,4], [5,5]	[sin,tanh], [sin,relu], [sin,sigmoid], [sin,sin]
3	[1,2,3], [2,2,2], [4,4,4] , [4,3,4], [5,5,5], [3,3,3]	[sin,tanh,tanh], [sin,relu,relu] , [sin,sigmoid,sigmoid], [sin, sin, sin]

Table 4: Possible values of the neural network structural parameters tried in the grid search. For multiple hidden layers, multiple values of hidden neurons and activation functions indicate the value for each layer. For example, if the number of hidden neurons is [4,3,4] and the activation functions are [sin, relu, relu], then the first hidden layer has 4 neurons with sin activation function, the second one 3 neurons with relu and the last one 4 neurons with relu.

with the contraint that the total number of weights is smaller than 55 (arbitrary choice to obtain a relevant compression).

Learning details

Every model was trained using Adam (Adaptive Moment Estimation)[6] with BackPropagation.

[1] An iteration through different combinations of them, performing the learning procedure described above on each one and then choosing as best the one which provides the best approximation performances.

While the basic Stochastic Gradient Descent update rule for the weight i, j in layer t at learning iteration s is (Eq. 22, omitting weight positional indices for ease of notation):

$$w^{(s+1)} = w^{(s)} - \eta \frac{\partial L_s}{\partial w^{(s)}} \tag{64}$$

$$\frac{\partial L_s}{\partial w^{(s)}} \equiv g^{(s)} \tag{65}$$

, Adam computes adaptive learning rates for each weight.
It stores an exponentially decaying average of past squared gradients $v^{(s)}$ and gradients $m^{(s)}$, similar to momentum.
Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which then prefers flat minima in the error surface [4].
$m^{(s)}$ and $v^{(s)}$ are computed as follows:

$$m^{(s)} = \beta_1 m^{(s-1)} + (1 - \beta_1) g^{(s)} \tag{66}$$

$$v^{(s)} = \beta_2 v^{(s-1)} + (1 - \beta_2)(g^{(s)})^2 \tag{67}$$

where β_1 and β_2 are the decay rates parameters.
$m^{(s)}$ and $v^{(s)}$ are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.
As all the $m^{(s)}$ and $v^{(s)}$ are initialized as vectors of zeros, they are biased towards zero, especially during the initial time steps and when β_1 and β_2 are close to 1.
To counteract this, bias-corrected first and second moment estimates are computed:

$$\hat{m}^{(s)} = \frac{m^{(s)}}{1 - \beta_1^s} \tag{68}$$

$$\hat{v}^{(s)} = \frac{v^{(s)}}{1 - \beta_2^s} \tag{69}$$

Then the Adam update rule reads:

$$w^{(s+1)} = w^{(s)} - \frac{\eta}{\sqrt{\hat{v}^{(s)}} + \epsilon} \hat{m}^{(s)} \tag{70}$$

Proposed default values for the Adam parameters are:

$$\beta_1 = 0.6, \qquad \beta_2 = 0.999, \qquad \epsilon = 10^{-8}$$

These are used in this work.

The maximum number of learning iterations τ is set to 10 Milions, but with an EarlyStopping schedule: if the loss doesn't decrease for a certain number of iterations (set to 5000), the learning is stopped.

The initial Learning Rate η was set to 0.01. To obtain better performances, if during training the loss stops improving (during, in this work, 500 iterations), then it is reduced by a factor 0.8.

FFNNs Results

The configurations of parameters that provided the lowest RMS for every number of hidden layers tried are reported in Table 5.
Here, it is easy to see how these performances are nowhere near acceptable.

Number of Hidden Layers	N	Activation Functions	Q	RMS [m]
1	10	sin	31	1170.665
1	12	sin	37	2987.542
1	13	sin	40	4186.150
2	[5, 5]	[sin, sin]	46	3444.291
2	[4, 4]	[sin, sin]	33	7642.957
2	[3, 3]	[sin, sin]	22	10921.536
3	[4, 3, 4]	[sin, sin, sin]	44	3660.580
3	[4, 4, 4]	[sin, sin, sin]	53	8608.745
3	[3, 3, 3]	[sin, sin, sin]	44	12457.375

Table 5: FFNN Best grid search results. N indicates the number of hidden neurons per layer and Q the total number of coefficients required (weights + biases).

Extreme Learning Machines

Extreme learning machines require just few structural parameters to be tuned: the activation functions of the hidden neurons and their number N.
However, one has to choose the weights going from the input layer to the hidden one $\boldsymbol{W}^{(1)}$. These are usually sampled from a normal or uniform distribution:

$$\text{Normal Distribution PDF: } \mathcal{N}(\mu,\, \sigma^2)(x) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{(x-\mu)^2}{2\sigma^2} \right) \tag{71}$$

$$\text{Uniform Distribution PDF: } \mathcal{U}(a,b)(x) = \begin{cases} \dfrac{1}{b-a}, & x \in [a,b] \\ 0, & \text{otherwise} \end{cases} \tag{72}$$

which are parametric: the former requires mean and standard deviation μ, σ, while the latter the extrema a, b.

These, together with the other parameters, even in this case, are chosen through a grid search.

Grid Search

The possible values of such parameters tried are:

Number of Hidden Neurons (N)	Input Weights & Biases sampling	Activation Functions
15,16,17,...,34	$\mathcal{N}(0,\, N^2//4)$, $\mathcal{N}(0,\, N^2)$, $\mathcal{N}(0,\, N^2//16)$, $\mathcal{U}(-N, N)$, $\mathcal{U}(-N//2, N//2)$, $\mathcal{U}(-N//4, N//4)$	Sigmoid, ReLU ,Sin, Tanh, Gaussian RBF

Table 6: Possible values of the ELM parameters tried during the grid search. The input weights and biases sampling indicates the random distribution $\boldsymbol{W}^{(1)}$ and $\boldsymbol{b}^{(1)}$ are drawn from.

The // operator indicates the floor division.

ELMs Results

Given the randomness of the input layer biases and weights, the training procedure of every combination of the parameters in Table 6 is repeated 10000 times, in order to obtain more "refined" results. Then, the RMS reported in Table 7 is the lowest one obtained during these 10000 iterations.

N	Input Weights & Biases sampling	Activation Functions	Q	RMS [m]
15	$\mathcal{N}(0,\, N^2//4)$	sin	45	110.224
15	$\mathcal{U}(-N//2,\, N//2)$	tanh	45	1019.189
16	$\mathcal{N}(0,\, N^2//4)$	sin	48	34.232
16	$\mathcal{U}(-N//2,\, N//2)$	sigmoid	48	500.293
17	$\mathcal{N}(0,\, N^2//4)$	sin	51	35.203
17	$\mathcal{U}(-N//2,\, N//2)$	tanh	51	430.117
18	$\mathcal{N}(0,\, N^2//4)$	sin	54	18.726
18	$\mathcal{U}(-N//2,\, N//2)$	sin	54	149.565
19	$\mathcal{N}(0,\, N^2//4)$	sin	57	9.255
19	$\mathcal{U}(-N//2,\, N//2)$	sin	57	100.415
20	$\mathcal{N}(0,\, N^2//4)$	sin	60	3.278
20	$\mathcal{U}(-N//2,\, N//2)$	sin	60	51.673
21	$\mathcal{N}(0,\, N^2//4)$	sin	63	0.846
21	$\mathcal{U}(-N//2,\, N//2)$	sin	63	19.896
22	$\mathcal{N}(0,\, N^2//4)$	sin	66	0.841
22	$\mathcal{U}(-N//2,\, N//2)$	sin	66	15.020
23	$\mathcal{N}(0,\, N^2//4)$	sin	69	0.218
23	$\mathcal{U}(-N//2,\, N//2)$	sin	69	13.736
24	$\mathcal{N}(0,\, N^2//4)$	sin	72	0.214
24	$\mathcal{U}(-N//2,\, N//2)$	sin	72	11.712
25	$\mathcal{N}(0,\, N^2//4)$	sin	75	0.0405
25	$\mathcal{U}(-N//2,\, N//2)$	sin	75	4.200
26	$\mathcal{N}(0,\, N^2//4)$	sin	78	0.036
26	$\mathcal{U}(-N//2,\, N//2)$	sin	78	4.344
27	$\mathcal{N}(0,\, N^2//4)$	sin	81	0.0253
27	$\mathcal{U}(-N//2,\, N//2)$	sin	81	3.564
28	$\mathcal{N}(0,\, N^2//4)$	sin	84	0.0242
28	$\mathcal{U}(-N//2,\, N//2)$	sigmoid	84	3.033
29	$\mathcal{N}(0,\, N^2//4)$	sin	87	0.0252
29	$\mathcal{U}(-N//2,\, N//2)$	sin	87	6.634
30	$\mathcal{N}(0,\, N^2//4)$	sin	90	0.022
30	$\mathcal{U}(-N//2,\, N//2)$	sin	90	6.129
31	$\mathcal{N}(0,\, N^2//4)$	sin	93	0.0219
31	$\mathcal{U}(-N//2,\, N//2)$	sigmoid	93	3.621
32	$\mathcal{N}(0,\, N^2//4)$	sin	96	0.0246
32	$\mathcal{U}(-N//2,\, N//2)$	sin	96	4.410
33	$\mathcal{N}(0,\, N^2//4)$	sin	99	0.033
33	$\mathcal{U}(-N//2,\, N//2)$	sigmoid	99	1.857
34	$\mathcal{N}(0,\, N^2//4)$	sin	102	0.0235
34	$\mathcal{U}(-N//2,\, N//2)$	sin	102	8.332

Table 7: ELM Grid search best results. N indicates the number of neurons in the hidden layer, and Q the number of coefficients required (which are $\boldsymbol{W}^{(1)}$, $\boldsymbol{b}^{(1)}$, $\boldsymbol{W}^{(2)}$), i.e. $3 \times N$.
For every N, are reported the configurations of parameters that provided the two lowest RMS; in any case, and for several re-runs, the best input weights and biases sampling distribution was the normal one, and the activation funcion the sine. The // operator indicates the floor division.

The coefficients required to reconstruct the x-coordinate are $\boldsymbol{W}^{(1)}$, $\boldsymbol{b}^{(1)}$, $\boldsymbol{W}^{(2)}$.
However, since both $\boldsymbol{W}^{(1)}$, $\boldsymbol{b}^{(1)}$ are pseudo-randomly generated, they can be reobtained by fixing the random seed: a pseudorandom number generator's number sequence is completely determined by the seed; thus, if a pseudorandom number generator is reinitialized with the same seed, it will produce the same sequence of numbers, independently of the machine it is run on.

In this work, normal random numbers are generated using the Ziggurat algorithm [14].

So, for every number of hidden neurons, one might just repeat the sampling with many random seeds, compute the output weights $\boldsymbol{W}^{(2)}$ and then store the 2 random seeds (one for the weights, one for the biases) that provided the lowest RMS.
In this way, the total number of coefficients to be saved is only $N + 2$.

The best results (Table 7) were *consinstently* obtained using a sinusoidal activation function and normally distributed input weights and biases.
In this way, the output of the machine $\tilde{x}$ when a time t is fed to it is a linear combination of sines:

$$\tilde{x} = w_{1,0}^{(2)} \sin\left(w_{0,1}^{(1)} t + b_1\right) + w_{2,0}^{(2)} \sin\left(w_{0,2}^{(1)} t + b_2\right) + \cdots + w_{N,0}^{(2)} \sin\left(w_{0,N}^{(1)} t + b_N\right) \tag{73}$$

Renaming, for easier notation:

$$w_{1,0}^{(2)}, w_{2,0}^{(2)}, \cdots, w_{N,0}^{(2)} \quad \longrightarrow \quad A_1, A_2, \cdots A_N \tag{74}$$

$$w_{0,1}^{(1)}, w_{0,2}^{(1)}, \cdots, w_{0,N}^{(1)} \quad \longrightarrow \quad \omega_1, \omega_2, \cdots, \omega_N \tag{75}$$

$$b_1, b_2, \cdots, b_N \quad \longrightarrow \quad \phi_1, \phi_2, \cdots, \phi_N \tag{76}$$

one gets:

$$\tilde{x} = A_1 \sin\left(\omega_1 t + \phi_1\right) + A_2 \sin\left(\omega_2 t + \phi_2\right) + \cdots + A_N \sin\left(\omega_N t + \phi_N\right) \tag{77}$$

$$= \sum_{k=1}^{N} A_k \sin\left(w_k t + \phi_k\right) \tag{78}$$

where the ω_k and ϕ_k, $k = 1, \cdots, N$ are randomly generated according to a Normal Distribution with mean zero and standard deviation equal to the number of hidden neurons N, and the A_k are computed using Eq. 35.

Given the form of Eq. 78, it is possible to refer to A_k, ω_k and ϕ_k as *amplitudes*, *frequencies* and *phases*, respectively.

Thus, the algorithm proposed to perform a one-day, one-coordinate ephemeris compression using "Extreme Learning Machines" with hidden layer size N is:

Algorithm 4 ELM ephemeris compression

function ELM REGRESSION(data points,N, τ)
    data points: $(t_1, x_1), (t_2, x_2), .., (t_m, x_m)$
    N: number of hidden layer nodes
    τ: number of random trials

    for $s = 1, 2, ..., \tau$ **do**
        set the input weights random seed $\mathcal{S}_\omega$
            ↳ draw N random input weights $\omega_k \sim \mathcal{N}(0,\ N^2//4)$
        set the input bias random seed $\mathcal{S}_\phi$
            ↳ draw N random input biases $\phi_k \sim \mathcal{N}(0,\ N^2//4)$
        Compute the matrix H

$$H = \begin{pmatrix} \sin(\omega_1 t_1 + \phi_1) & \cdots & \sin(\omega_N t_1 + \phi_N) \\ \sin(\omega_1 t_2 + \phi_1) & \cdots & \sin(\omega_N t_2 + \phi_N) \\ \vdots & \ddots & \vdots \\ \sin(\omega_1 t_m + \phi_1) & \cdots & \sin(\omega_N t_m + \phi_N) \end{pmatrix}$$

        compute the amplitudes $\boldsymbol{A} = (A_1, \cdots, A_N)^T$ (H^+ is the generalized inverse of H):

$$\boldsymbol{A} = H^+ \boldsymbol{X} \qquad \text{with } \boldsymbol{X} \equiv (x_1, x_2, \cdots, x_m)^T$$

        compute the sum of squared residuals:

$$\mathcal{L} = \|H\boldsymbol{A} - \boldsymbol{X}\|^2$$

    end for

    return $\boldsymbol{A}$, $\mathcal{S}_\omega$, $\mathcal{S}_\phi$ that provided the smallest $\mathcal{L}$

This ELM learning procedure was repeated τ =1 Milion times for every number of hidden neurons $N \in [18, 35]$. The results are reported in Table 8.

Alternatively, one might perform the ELM regression according to the *Reformulation of ELM Learning as Least Squares Regression*.

A new metric is introduced from now on, a sort of Compression Ratio, CR[2]:

$$CR \equiv \frac{\text{Number of Data Points}}{\text{Number of Regression Coefficient Required}} = \frac{m}{Q} \tag{79}$$

For ELM regression, the Number of Regression Coefficients Q is $N + 2$, while for polynomial regression is Polynomial Degree +1.

[2]This is not the *actual* data compression ratio, which is defined as the ration between the uncompressed data size and the compressed data size.

Number of Hidden Neurons	Input Weights Seed	Input Biases Seed	Q	CR	RMS [m]	Max Error [m]
35	856790804	904304051	37	2.595	0.02	0.071
34	327596683	748514280	36	2.667	0.02	0.07
33	243932085	256261745	35	2.743	0.02	0.067
32	868534036	17240660	34	2.824	0.02	0.07
31	235120639	408581275	33	2.909	0.02	0.07
30	45471254	626179072	32	3.0	0.02	0.071
29	443566627	543027330	31	3.097	0.02	0.069
28	382600566	258666930	30	3.2	0.021	0.075
27	756513439	573390607	29	3.31	0.021	0.076
26	306643683	648382576	28	3.429	0.025	0.074
25	7424024	824524369	27	3.556	0.035	0.102
24	397058180	993157481	26	3.692	0.058	0.159
23	88783600	8481978	25	3.84	0.158	0.289
22	1665822207	948449523	24	4.0	0.324	1.041
21	23419336	25674658	23	4.174	0.638	2.115
20	712841537	719600262	22	4.364	1.629	6.059
19	32773900	6620175	21	4.571	4.356	10.942
18	571067917	715355113	20	4.8	8.37	39.703

Table 8: ELM with different number of hidden neurons N, sine activation function and normally distributed with mean zero and standard deviation $N//2$ weights and biases results, ordered by RMS. Q indicates the number of coefficients to be stored, i.e. $N + 2$ (only the output weights $\boldsymbol{W}^{(2)} \equiv \boldsymbol{A}$ and the two random seeds are required).
The compression ratio CR is given by Eq. 79 $CR = 96/Q$.
Note that, contrarily as one might expect, more hidden neurons don't result automatically in a lower RMS. This is easily explained with the randomness of the choice of the input weights and biases.

A comparison of the results obtained with polynomial regression and these ELM, with the same number of coefficients required is reported in Table 9 and shown in Fig. 4.

Q	CR	RMS [m]		Max Error [m]	
		Polynomial Regression	ELM	Polynomial Regression	ELM
20	4.8	423.574	8.37	1132.395	39.703
21	4.571	200.239	4.356	453.434	10.942
22	4.364	77.626	1.629	201.811	6.059
23	4.174	48.861	0.638	105.147	2.115
24	4.0	13.957	0.324	41.279	1.041
25	3.84	11.989	0.158	24.963	0.289
26	3.692	2.804	0.058	6.166	0.159
27	3.556	2.801	0.035	5.831	0.102
28	3.429	0.695	0.025	1.519	0.074
29	3.31	0.613	0.021	1.412	0.076
30	3.2	0.209	0.021	0.529	0.075
31	3.097	0.129	0.02	0.343	0.069
32	3.0	0.065	0.02	0.147	0.071
33	2.909	0.031	0.02	0.094	0.07
34	2.824	0.026	0.02	0.076	0.07
35	2.743	0.021	0.02	0.076	0.067
36	2.667	0.021	0.02	0.075	0.07
37	2.595	0.02	0.02	0.071	0.071

Table 9: Comparison between polynomial least squares regression and ELM with N neurons in the hidden layer with sine activation function and input weight and biases $\sim \mathcal{N}(0,\, N^2//4)$. Q,as usual, indicates the number of coefficients required. For polynomial regression it is 1+ polynomial degree, for ELM, 2+ number of hidden neurons. $CR = 96/Q$.

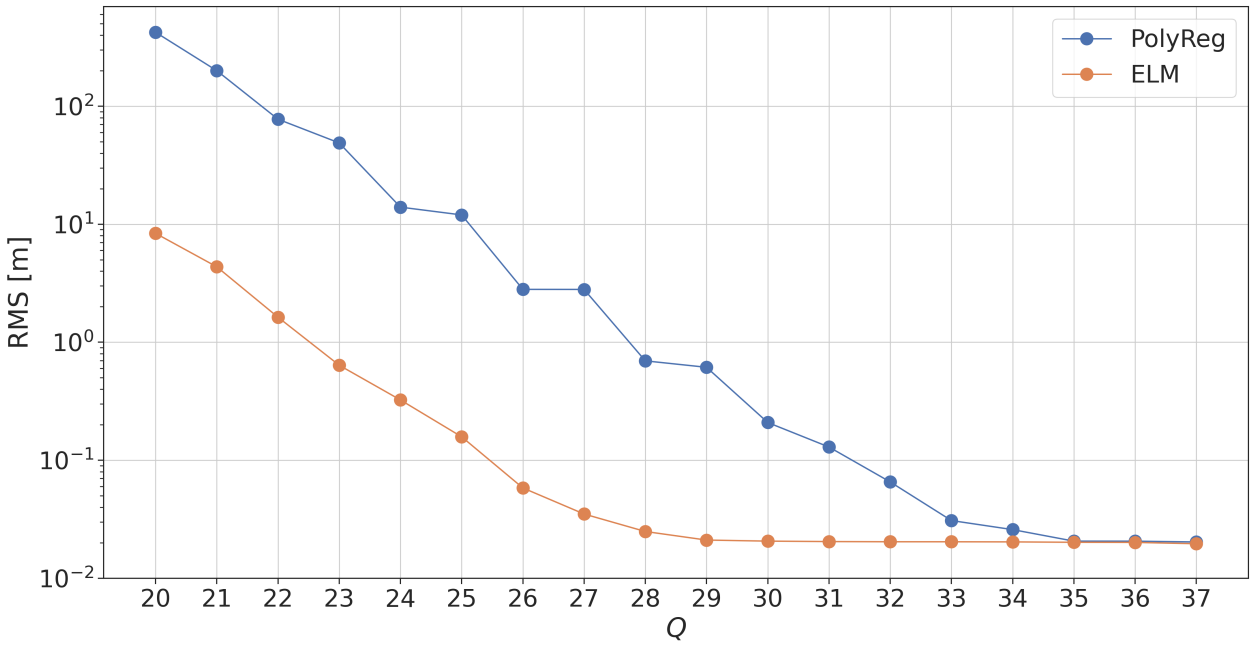

Figure 4: RMS as function of the number of coefficients required Q for polynomial least squares regression and ELM with sine activation function.

Given the good results of ELM regression with sine activation function and normally distributed (with mean zero and standard deviation equal to the floor division of the number of neurons in the hidden layer by two) input weights and biases (from now on in this section, simply refered to as ELM regression), the procedure described in Algorithm 4 was repeated also for 2, 5 and 7 days of data and $\tau = 100\,000$.

2 Days of Data

Two days of data correspond to $m = 192$ data points: (t_i, x_i).

The results obtained with ELMs are reported in Table 10.
A comparison with polynomial regression is made in Table 11 and shown in Fig. 5.

Number of Hidden Neurons	Input Weights Seed	Input Biases Seed	Q	CR	RMS [m]	Max Error [m]
66	28759813	38196729	68	2.824	0.021	0.071
68	93093898	83472150	70	2.743	0.021	0.07
64	87858726	31181540	66	2.909	0.021	0.072
62	80431831	76955296	64	3.0	0.021	0.072
58	45520112	76970661	60	3.2	0.021	0.071
60	73388700	11186482	62	3.097	0.021	0.076
56	99561959	80031759	58	3.31	0.021	0.077
54	34796748	87887063	56	3.429	0.022	0.078
52	94270509	69835155	54	3.556	0.022	0.078
50	79221488	38791408	52	3.692	0.025	0.092
48	59694104	274407	50	3.84	0.026	0.084
46	41394825	94662759	48	4.0	0.041	0.117
44	61294095	48519708	46	4.174	0.089	0.216
42	75834447	7797028	44	4.364	0.337	0.696
40	51081132	1748981	42	4.571	1.049	2.488
38	12165798	30947180	40	4.8	1.288	2.598
36	34508117	81877270	38	5.053	1.595	4.093
34	60019744	78051080	36	5.333	6.957	13.833
32	40177488	89750104	34	5.647	11.956	28.248
30	37500577	17530573	32	6.0	64.951	225.297

Table 10: ELM regression with different number of hidden neurons N results for 2 days of data, ordered by RMS.
Q indicates the number coefficients to be stored, i.e. $N + 2$ and $CR = 192/Q$.

Q	CR	RMS [m]		Max Error [m]	
		Polynomial Regression	ELM	Polynomial Regression	ELM
32	6.0	4248.343	64.951	10073.927	225.297
34	5.647	1655.77	11.956	4663.261	28.248
36	5.333	546.718	6.957	1574.561	13.833
38	5.053	186.64	1.595	469.87	4.093
40	4.8	92.917	1.288	185.647	2.598
42	4.571	53.512	1.049	100.109	2.488
44	4.364	26.659	0.337	52.716	0.696
46	4.174	11.132	0.089	22.543	0.216
48	4.0	4.175	0.041	8.246	0.117
50	3.84	1.744	0.026	3.63	0.084
52	3.692	1.022	0.025	2.185	0.092
54	3.556	0.66	0.022	1.333	0.078
56	3.429	0.376	0.022	0.754	0.078
58	3.31	0.183	0.021	0.395	0.077
60	3.2	0.08	0.021	0.179	0.071
62	3.097	0.037	0.021	0.105	0.076
64	3.0	0.026	0.021	0.085	0.072
66	2.909	0.022	0.021	0.074	0.072
68	2.824	0.021	0.021	0.074	0.071
70	2.743	0.021	0.021	0.075	0.07

Table 11: Comparison between polynomial least squares and ELM regression for 2 days of data. Q indicates the number of coefficients required and $CR = 192/Q$.

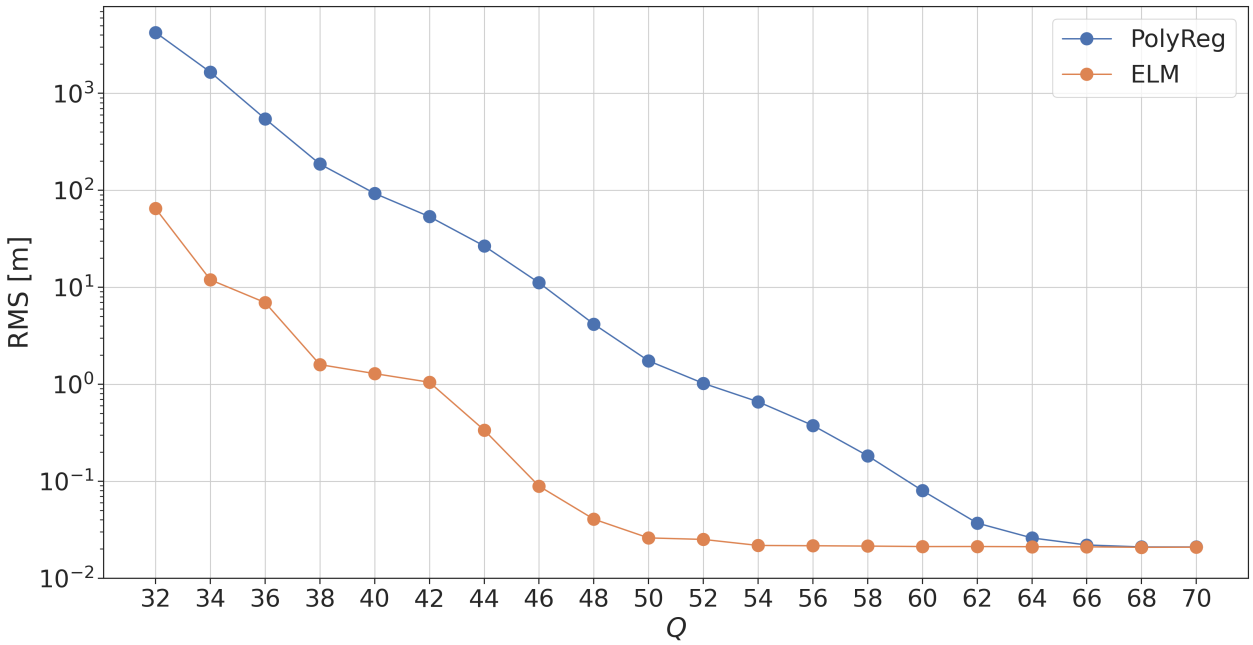

Figure 5: 2 days of data approximation RMS as function of the number of coefficients required Q for polynomial least squares and ELM regression.

5 Days of Data

Five days of data correspond to $m = 480$ data points: (t_i, x_i).
The results obtained with ELMs are reported in Table 12.
A comparison of these ELMs with polynomial regression, with the same number of coefficients required is reported in Table 13 and shown in Fig. 6.

Number of Hidden Neurons	Input Weights Seed	Input Biases Seed	Q	CR	RMS [m]	Max Error [m]
174	1681222	11626278	176	2.727	0.017	0.069
172	11019738	66384173	174	2.759	0.017	0.066
168	93337553	13504037	170	2.824	0.018	0.068
166	72722878	28384101	168	2.857	0.018	0.065
170	43976919	29827884	172	2.791	0.018	0.067
162	50337421	65717267	164	2.927	0.018	0.064
160	55750376	98917965	162	2.963	0.018	0.067
164	19260511	62762565	166	2.892	0.018	0.063
158	41115157	12846861	160	3.0	0.018	0.064
154	38119901	23475712	156	3.077	0.018	0.065
156	17475971	43734834	158	3.038	0.018	0.067
152	88630176	81964568	154	3.117	0.018	0.069
150	5936696	56100911	152	3.158	0.019	0.068
146	92650760	81641263	148	3.243	0.019	0.076
144	54067574	26000734	146	3.288	0.019	0.081
134	83278947	92647759	136	3.529	0.019	0.082
148	12876577	89081864	150	3.2	0.02	0.073
140	62761179	25199079	142	3.38	0.021	0.083
142	6112050	67972659	144	3.333	0.024	0.083
136	31306973	131967	138	3.478	0.024	0.087
138	12924896	14693709	140	3.429	0.024	0.086
132	427487289	241166900	134	3.582	0.03	0.117
128	94277903	86977696	130	3.692	0.032	0.111
126	50435001	47902254	128	3.75	0.033	0.121
130	6078693	94496914	132	3.636	0.042	0.124
124	44727754	97452749	126	3.81	0.05	0.138
114	234982613	309920319	116	4.138	0.076	0.251
120	872212773	341404053	122	3.934	0.1	0.322
122	344177473	906776975	124	3.871	0.107	0.286
118	98881385	46665048	120	4.0	0.117	0.345
116	210643663	279797827	118	4.068	0.157	0.475
112	677133807	861682471	114	4.211	0.254	0.705
110	365394512	665272058	112	4.286	0.672	1.339
108	30996812	60464896	110	4.364	0.776	1.859
106	43181696	72867539	108	4.444	1.009	2.341
104	183439591	103212185	106	4.528	1.197	2.126
102	39038172	8199728	104	4.615	1.249	2.431
98	28941135	8979757	100	4.8	1.254	2.198
100	811667008	360274012	102	4.706	1.295	2.957
94	75049467	34869358	96	5.0	2.948	10.198
92	11322377	89793510	94	5.106	3.645	12.518
96	5212899	89793510	98	4.898	6.067	16.125
90	705818	29097074	92	5.217	10.907	35.56

Table 12: ELM regression with different number of hidden neurons N results for 5 days of data, ordered by RMS.
Q indicates the number coefficients to be stored, i.e. $N + 2$ and $CR = 480/Q$.

Q	CR	RMS [m]		Max Error [m]	
		Polynomial Regression	ELM	Polynomial Regression	ELM
92	5.217	93.934	10.907	205.53	35.56
94	5.106	92.857	3.645	220.775	12.518
96	5.0	90.49	2.948	203.212	10.198
98	4.898	83.803	6.067	211.731	16.125
100	4.8	81.826	1.254	231.38	2.198
102	4.706	80.717	1.295	217.512	2.957
104	4.615	71.745	1.249	176.763	2.431
106	4.528	55.519	1.197	125.69	2.126
108	4.444	37.636	1.009	79.338	2.341
110	4.364	22.591	0.776	52.868	1.859
112	4.286	12.093	0.672	29.6	1.339
114	4.211	5.879	0.254	15.225	0.705
116	4.138	2.791	0.076	6.97	0.251
118	4.068	1.527	0.157	3.986	0.475
120	4.0	1.083	0.117	2.648	0.345
122	3.934	0.92	0.1	2.078	0.322
124	3.871	0.895	0.107	2.251	0.286
126	3.81	0.882	0.05	2.088	0.138
128	3.75	0.821	0.033	2.084	0.121
130	3.692	0.788	0.032	2.314	0.111
132	3.636	0.787	0.042	2.294	0.124
134	3.582	0.741	0.03	1.994	0.117
136	3.529	0.621	0.019	1.538	0.082
138	3.478	0.46	0.024	1.064	0.087
140	3.429	0.304	0.024	0.678	0.086
142	3.38	0.182	0.021	0.4	0.083
144	3.333	0.102	0.024	0.238	0.083
146	3.288	0.057	0.019	0.151	0.081
148	3.243	0.035	0.019	0.107	0.076
150	3.2	0.025	0.02	0.09	0.073
152	3.158	0.022	0.019	0.079	0.068
154	3.117	0.021	0.018	0.076	0.069
156	3.077	0.021	0.018	0.074	0.065
158	3.038	0.021	0.018	0.073	0.067
160	3.0	0.021	0.018	0.075	0.064
162	2.963	0.021	0.018	0.073	0.067
164	2.927	0.02	0.018	0.073	0.064
166	2.892	0.019	0.018	0.068	0.063
168	2.857	0.018	0.018	0.066	0.065
170	2.824	0.018	0.018	0.067	0.068
172	2.791	0.017	0.018	0.07	0.067
174	2.759	0.017	0.017	0.067	0.066
176	2.727	0.017	0.017	0.066	0.069

Table 13: Comparison between polynomial least squares and ELM regression for 5 days of data and different number of coefficients required Q. For polynomial regression it is 1+ polynomial degree, for ELM, 2+ number of hidden neurons. $CR = 480/Q$.

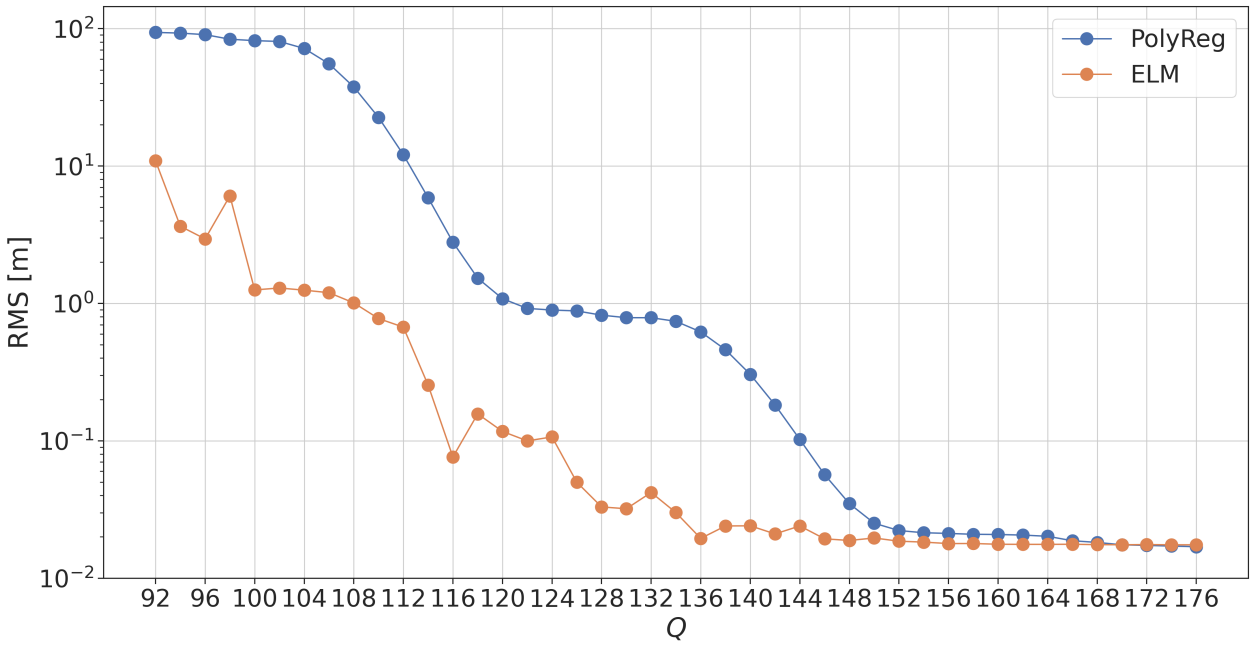

Figure 6: 5 days of data approximation RMS as function of the number of coefficients required Q for polynomial least squares and ELM regression.

7 Days of Data

7 days of data correspond to $m = 672$ data points: (t_i, x_i).

The results obtained with ELMs are reported in Table 14.
A comparison with polynomial regression is made in Table 15 and shown in Fig. 7.

Number of Hidden Neurons	Input Weights Seed	Input Biases Seed	Q	CR	RMS [m]	Max Error [m]
224	75934645	5772961	226	2.973	0.02	0.071
214	29314503	9313270	216	3.111	0.02	0.07
222	98588305	98584296	224	3.0	0.02	0.072
220	7613739	61813145	222	3.027	0.02	0.074
212	72926379	75587007	214	3.14	0.02	0.074
218	52431100	3410522	220	3.055	0.02	0.075
210	1767399	26603339	212	3.17	0.021	0.077
216	77414066	53909551	218	3.083	0.021	0.079
206	71075979	15566364	208	3.231	0.021	0.078
194	5341533	19137590	196	3.429	0.022	0.083
208	43730738	771050	210	3.2	0.023	0.08
204	89910191	45843020	206	3.262	0.024	0.078
202	73850630	51302578	204	3.294	0.024	0.087
200	346979565	22066784	202	3.327	0.026	0.086
198	26497097	98954450	200	3.36	0.026	0.087
190	89334703	23124668	192	3.5	0.026	0.094
196	57660769	70135715	198	3.394	0.028	0.1
182	30213675	76365320	184	3.652	0.028	0.1
192	39644013	15839126	194	3.464	0.029	0.106
178	13027203	33677327	180	3.733	0.029	0.091
186	94050232	57757778	188	3.574	0.03	0.096
188	42598490	54779105	190	3.537	0.034	0.153
184	66872073	57204694	186	3.613	0.05	0.145
180	38075578	47468006	182	3.692	0.052	0.185
172	28309921	77094957	174	3.862	0.069	0.229
176	33471976	49908861	178	3.775	0.08	0.273
160	85004977	82450304	162	4.148	0.085	0.26
174	22864644	94204802	176	3.818	0.09	0.252
168	73765637	71272728	170	3.953	0.103	0.384
170	45916393	50862840	172	3.907	0.122	0.458
166	98994248	61730252	168	4.0	0.208	0.571
164	319813412	677161565	166	4.048	0.255	0.74
162	913516913	858754026	164	4.098	0.259	0.772
156	32341918	49383012	158	4.253	0.562	1.443
158	20343748	96319084	160	4.2	0.767	2.417
152	55690058	45404861	154	4.364	0.865	2.401
150	60812741	70496358	152	4.421	0.905	2.108
154	82923911	48480188	156	4.308	0.94	2.4
144	2821713	60299778	146	4.603	1.314	2.608
148	61894021	94968501	150	4.48	1.354	3.614
146	55409705	50741447	148	4.541	1.482	4.016
140	86163231	4298953	142	4.732	1.783	4.814
142	81518480	51147473	144	4.667	1.863	4.784
138	69375244	20119369	140	4.8	2.188	6.26
130	40591936	84863863	132	5.091	2.308	7.819
132	58533116	64088948	134	5.015	4.668	21.465
136	69381659	76854864	138	4.87	4.789	14.153
126	973623	41500857	128	5.25	5.513	16.297

Table 14: ELM regression with different number of hidden neurons N results for 7 days of data, ordered by RMS.
Q indicates the number coefficients to be stored, i.e. $N + 2$ and $CR = 672/Q$.

Q	CR	RMS [m]		Max Error [m]	
		Polynomial Regression	ELM	Polynomial Regression	ELM
126	5.333	98.035	17.638	216.799	52.368
128	5.25	94.995	5.513	211.609	16.297
130	5.169	94.278	7.142	218.706	17.607
132	5.091	92.524	2.308	206.911	7.819
134	5.015	88.562	4.668	213.202	21.465
136	4.941	87.938	7.065	219.724	17.163
138	4.87	86.088	4.789	206.693	14.153
140	4.8	80.447	2.188	205.651	6.26
142	4.732	77.257	1.783	230.491	4.814
144	4.667	77.109	1.863	229.619	4.784
146	4.603	73.282	1.314	202.577	2.608
148	4.541	62.857	1.482	159.803	4.016
150	4.48	48.285	1.354	114.071	3.614
152	4.421	33.432	0.905	74.466	2.108
154	4.364	21.003	0.865	52.078	2.401
156	4.308	12.077	0.94	32.173	2.4
158	4.253	6.501	0.562	17.254	1.443
160	4.2	3.427	0.767	8.95	2.417
162	4.148	1.872	0.085	4.938	0.26
164	4.098	1.156	0.259	2.902	0.772
166	4.048	0.953	0.255	2.42	0.74
168	4.0	0.93	0.208	2.178	0.571
170	3.953	0.905	0.103	2.14	0.384
172	3.907	0.903	0.122	2.208	0.458
174	3.862	0.886	0.069	2.033	0.229
176	3.818	0.851	0.09	2.161	0.252
178	3.775	0.846	0.08	2.271	0.273
180	3.733	0.834	0.029	2.106	0.091
182	3.692	0.784	0.052	1.987	0.185
184	3.652	0.744	0.028	2.268	0.1
186	3.613	0.741	0.05	2.35	0.145
188	3.574	0.728	0.03	2.19	0.096
190	3.537	0.662	0.034	1.844	0.153
192	3.5	0.547	0.026	1.413	0.094
194	3.464	0.41	0.029	0.999	0.106
196	3.429	0.28	0.022	0.665	0.083
198	3.394	0.178	0.028	0.423	0.1
200	3.36	0.107	0.026	0.26	0.087
202	3.327	0.062	0.026	0.176	0.086
204	3.294	0.038	0.024	0.136	0.087
206	3.262	0.027	0.024	0.109	0.078
208	3.231	0.024	0.021	0.097	0.078
210	3.2	0.023	0.023	0.09	0.08
212	3.17	0.023	0.021	0.085	0.077
214	3.14	0.022	0.02	0.081	0.074
216	3.111	0.022	0.02	0.079	0.07
218	3.083	0.022	0.021	0.079	0.079
220	3.055	0.022	0.02	0.078	0.075
222	3.027	0.022	0.02	0.077	0.074

Table 15: Comparison between polynomial least squares and ELM regression for 7 days of data. Q indicates the number of coefficients required. $CR = 672/Q$.

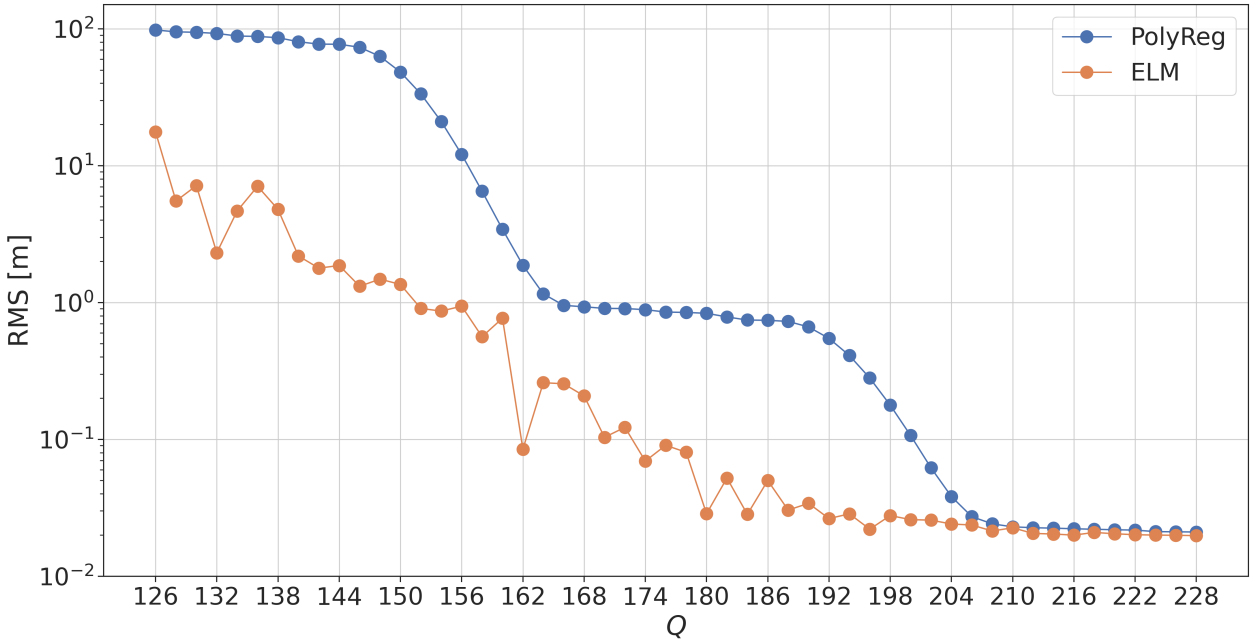

Figure 7: 7 days of data approximation RMS as function of the number of coefficients required Q for polynomial least squares and ELM regression.

Unweighted Regression - Summary

In Table 16 and Fig. 8 are summarized the minimum numbers of coefficients required in order to achieve an $RMS \leqslant 0.1m$, obtained via weighted ELM and Polynomial regression for different number of days of data.

$t_f - t_0$ (Days of Data)	$Q_{PolyReg}$	Q_{ELM}
1	32	26
2	60	46
5	145	116
7	201	162

Table 16: Summary of the minimum number of coefficients required in order to achieve an $RMS \leqslant 0.1m$, obtained via weighted ELM nd Polynomial regression (Q_{ELM} and $Q_{PolyReg}$ respectively) for different number of days of data.

The increase of Q with respect to the number of days of data is then approximated with a power law $Q \approx a(t_f - t_0)^b$ by performing a least square regression :

$$Q_{PolyReg} \approx 31.12(t_f - t_0)^{0.96} \implies Q_{PolyReg} \sim \mathcal{O}\big((t_f - t_0)^{0.96}\big)$$
$$Q_{ELM} \approx 24.02(t_f - t_0)^{0.98} \implies Q_{ELM} \sim \mathcal{O}\big((t_f - t_0)^{0.98}\big)$$

Clearly, for Q_{ELM} there is some uncertainty on the power law coefficients found, given the randomness of the procedure and the few $t_f - t_0$ tried.

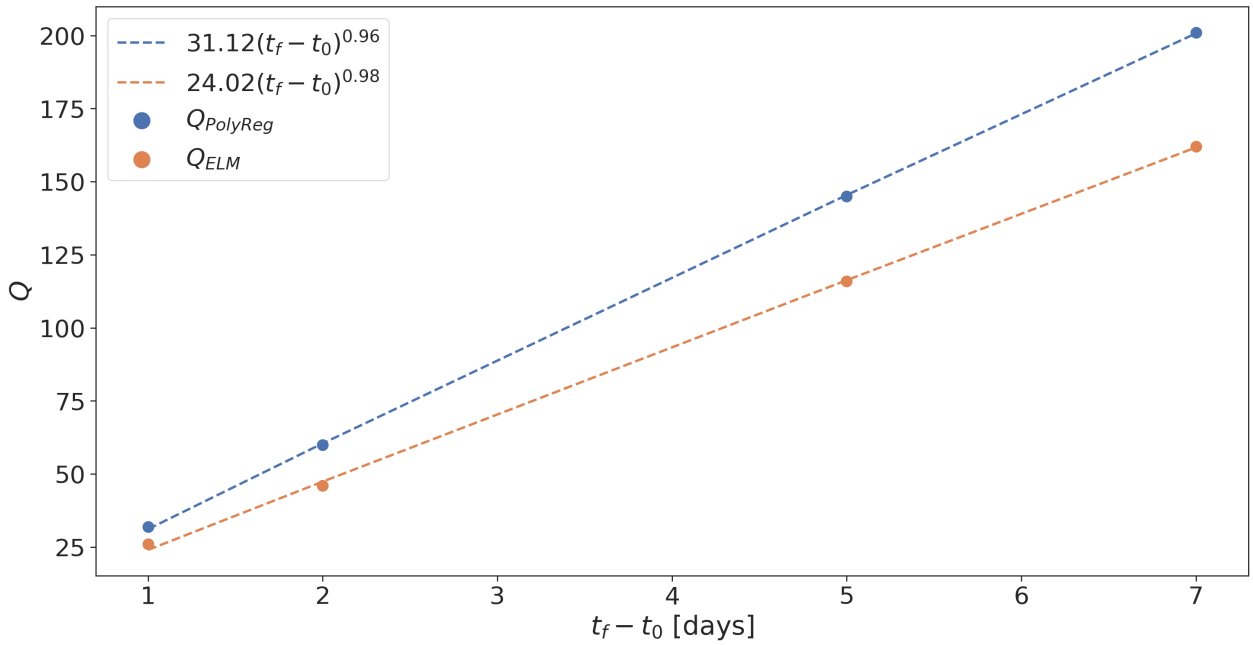

Figure 8: Minimum number of coefficients required obtained via ELM and Polynomial regression (Q_{ELM} and $Q_{PolyReg}$ respectively) for different number of days of data $t_f - t_0$.
The dashed lines represent a power law increase of $Q \approx a(t_f - t_0)^b$ with the best fit coefficients a, b.
The increase of both Q_{ELM} and $Q_{PolyReg}$ is sublinear.

III.2 Weighted Regression

As said before, the propagated orbits come with an error e (Eq. 1) with respect to the official igs ones.

This enables the possibility to perform a weighted regression, where the weight associated to every point is a decreasing function of the error of propagation e.

The main idea is to perform a regression on the propagated orbits, and to constrain the approximation error $l(t)$

$$l(t) = |\tilde{x}(t) - x(t)| \tag{80}$$

(where $\tilde{x}(t)$ and $x(t)$ indicate the reconstructed coordinate from the ELM coefficients found (Eq.52) and the propagated data ones, respectively) to be smaller than the propagation one ($e(t)$, Eq. 1) $\forall t$.

However, the propagation error is not used directly, but its linear parametrization $\tilde{e}$: the initial error $\tilde{e}(t_0)$ is set to $1cm$ and the final one is equal to the actual error at the end of the time span $e(t_f)$. The slope and intercept of the line is automatically chosen:

$$\tilde{e}(t) = \tilde{e}(t_0) + a(t - t_i) \tag{81}$$

$$a = \frac{e(t_f) - \tilde{e}(t_i)}{t_f - t_i} \tag{82}$$

The weights in Eqs.52,49 are set to be the reciprocal of the linear error of propagation (Eq.1) $w = 1/\tilde{e}$.

The regression strategy followed is:

1. set an initial Number of neurons in the hidden layer, N;
2. perform a weighted ELM regression to find the best coefficients (amplitudes, phases and frequencies);

3. Reconstruct the data given the coefficients found and compute its error with respect to the data points (Eq. 80);
4. If exists a t such that the reconstruction error $l(t)$ is larger than the error of propagation $\tilde{e}(t)$ beyond a certain tolerance (set by the user, means that $|l(t) - \tilde{e}(t)| \geqslant$ tolerance), then increase N;
5. Iterate until the reconstruction error $l(t)$ becomes smaller than the linear propagation one $\tilde{e}(t)$.

More specifically:

Algorithm 5 Weighted ELM ephemeris compression with error control

function WEIGHTED ELM REGRESSION(data points, data weights, error, η, τ)
    data points: $((t_1, x_1), (t_2, x_2), .., (t_m, x_m))$
    data weights: $(w_1, w_2, ..., w_m)$
    error: parametrized propagation error $\tilde{e} = (\tilde{e}_1, \tilde{e}_2, ..., \tilde{e}_m)$
    η: tolerance on error
    τ: number of random trials

    set an initial N
    while True **do**
        for $s = 1, 2, ..., \tau$ **do**
            set the input weights random seed $\mathcal{S}_\omega$
            ↳ draw N random input weights $\omega_k \sim \mathcal{N}(0,\, N^2//4)$
            set the input bias random seed $\mathcal{S}_\phi$
            ↳ draw N random input biases $\phi_k \sim \mathcal{N}(0,\, N^2//4)$
            Compute the matrix H

$$H = \begin{pmatrix} \sin(\omega_1 t_1 + \phi_1) & \cdots & \sin(\omega_N t_1 + \phi_N) \\ \sin(\omega_1 t_2 + \phi_1) & \cdots & \sin(\omega_N t_2 + \phi_N) \\ \vdots & \ddots & \vdots \\ \sin(\omega_1 t_m + \phi_1) & \cdots & \sin(\omega_N t_m + \phi_N) \end{pmatrix}$$

            Compute the matrix $\tilde{A}$ and vector $\boldsymbol{b}$

$$\hat{A} \equiv \sum_{i=1}^{m} w_i \boldsymbol{H_i}^T \boldsymbol{H_i}, \qquad \hat{A}_{jk} = \sum_{i=1}^{m} w_i H_{ij} H_{ik} \tag{83}$$

$$\boldsymbol{b} \equiv \sum_{i=1}^{m} w_i y_i \boldsymbol{H_i}^T \tag{84}$$

            Compute the amplitudes $\boldsymbol{A} = (A_1, \cdots, A_N)^T$:

$$\boldsymbol{A} = \hat{A}^{-1} \boldsymbol{b}\,, \qquad \boldsymbol{X} \equiv (x_1, x_2, \cdots, x_m)^T$$

            Compute the ELM reconstructed data $\tilde{\boldsymbol{X}}$ and the sum of squared residuals $\mathcal{L}$:

$$\tilde{\boldsymbol{X}} = H\boldsymbol{A}, \qquad \tilde{\boldsymbol{X}} = (\tilde{x}_1, ..., \tilde{x}_m)^T$$
$$\mathcal{L} = \|H\boldsymbol{A} - \boldsymbol{X}\|^2$$

        end for

        if exist $\mathcal{S}_\omega, \mathcal{S}_\phi$ such that $|x_i - \tilde{x_i}| < \tilde{e}_i + \eta \;\forall i$ **then**
            end while
            return $\boldsymbol{A}$, $\mathcal{S}_\omega$, $\mathcal{S}_\phi$ that provided the smallest $\mathcal{L}$
        else
            $N = N + 1$

A very similar procedure was elaborated also for a polynomial regression; the only difference is in Step 2, where in this case a weighted polynomial regression is performed (see section II.1 - *Weighted Least Squares Polynomial Regression*) (only once, given there is no randomness). In Step 1, the number of hidden neurons is replaced with the polynomial degree -1; Steps 3, 4 and 5 are exactly the same.

Here are reported the results obtained via weighted ELM (with $\tau = 100\,000$) and Polynomial Regression for different numbers of days of data: 1, 2, 5, 7, 10, 14.

1 Day of Data

A plot of the parametrization of the propagation error and its associated weight for 1 day of data are shown as function of time in Fig.9.

A comparison of the lowest number of coefficients required to satisfy the constraints on the approximation error between ELM and Polynomial weighted regression is shown in Table 17 and Fig. 10.

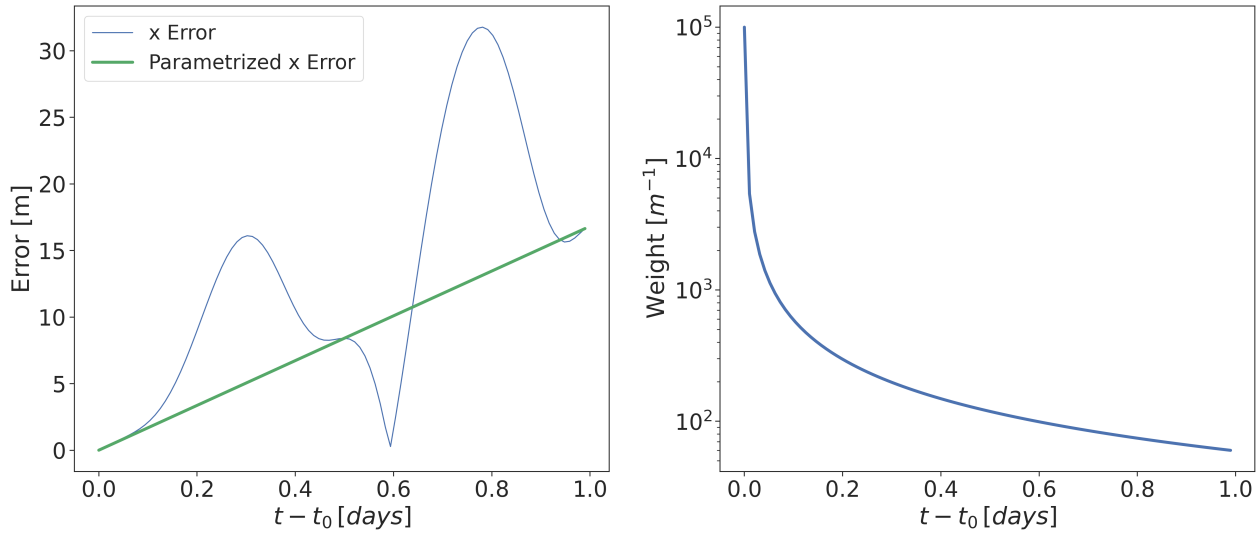

Figure 9: (**Left**) Propagation error and its linear parametrization for 1 day of data. (**Right**) Weight associated to the data points.

	Q	CR	Weights Seed	Biases Seed	RMS [m]	Max Error [m]
ELM	23	4.174	26922664	53405801	1.254	2.761
PolyReg	28	3.428	-	-	0.913	1.742

Table 17: Weighted ELM and Polynomial regression results for 1 day of data. $CR = 96/Q$.

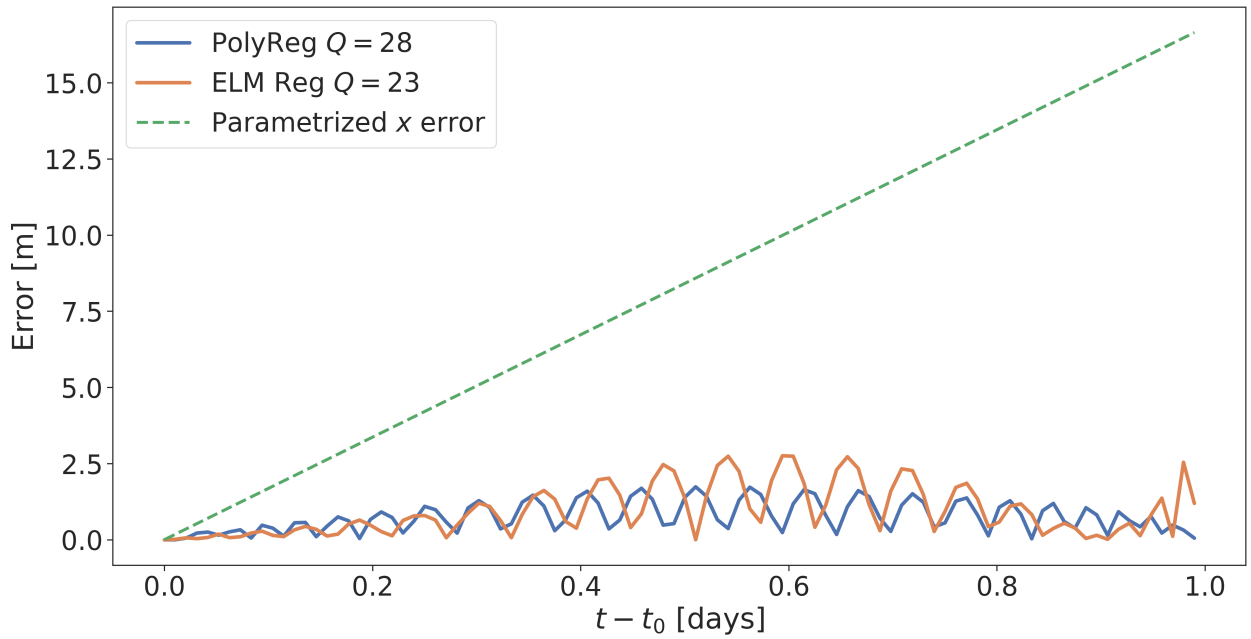

Figure 10: Weighted ELM and Polynomial regression approximation error $l(t)$ for 1 day of data.

2 Days of Data

A plot of the parametrization of the propagation error and its associated weight for 2 days of data are shown as function of time in Fig.11.

A comparison of the lowest number of coefficients required to satisfy the constraints on the approximation error between ELM and Polynomial weighted regression is shown in Table 18 and Fig. 12.

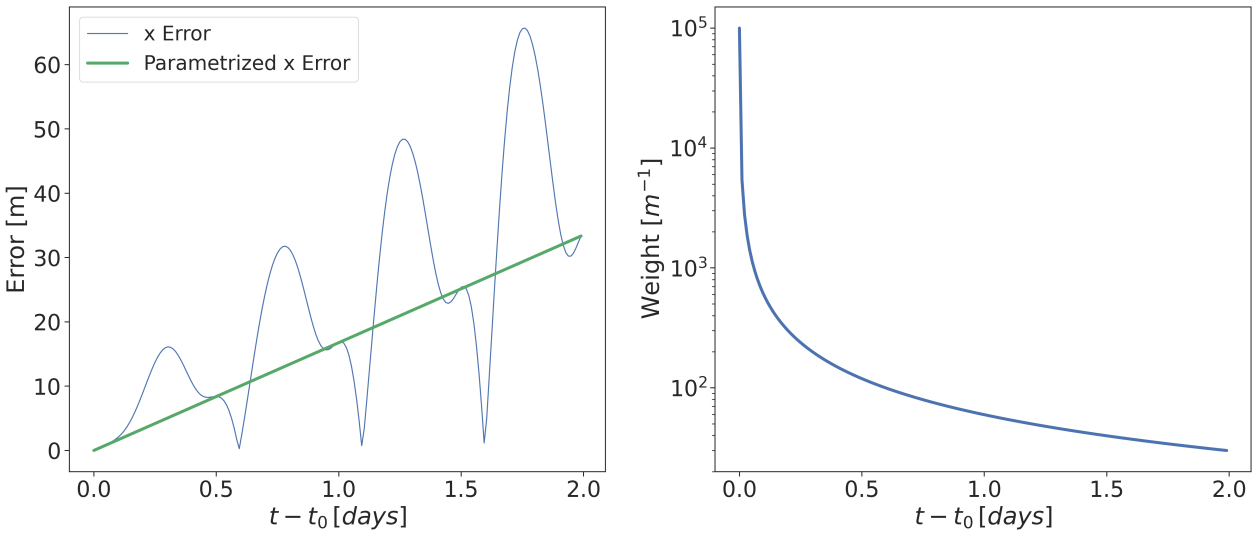

Figure 11: (**Left**) Propagation error and its linear parametrization for 2 days of data. (**Right**) Weight associated to the data points.

	Q	CR	Weights Seed	Biases Seed	RMS [m]	Max Error [m]
ELM	37	5.189	45563894	7104959	5.356	11.367
PolyReg	49	3.918	-	-	6.186	21.587

Table 18: Weighted ELM and Polynomial regression results for 2 days of data. $CR = 192/Q$.

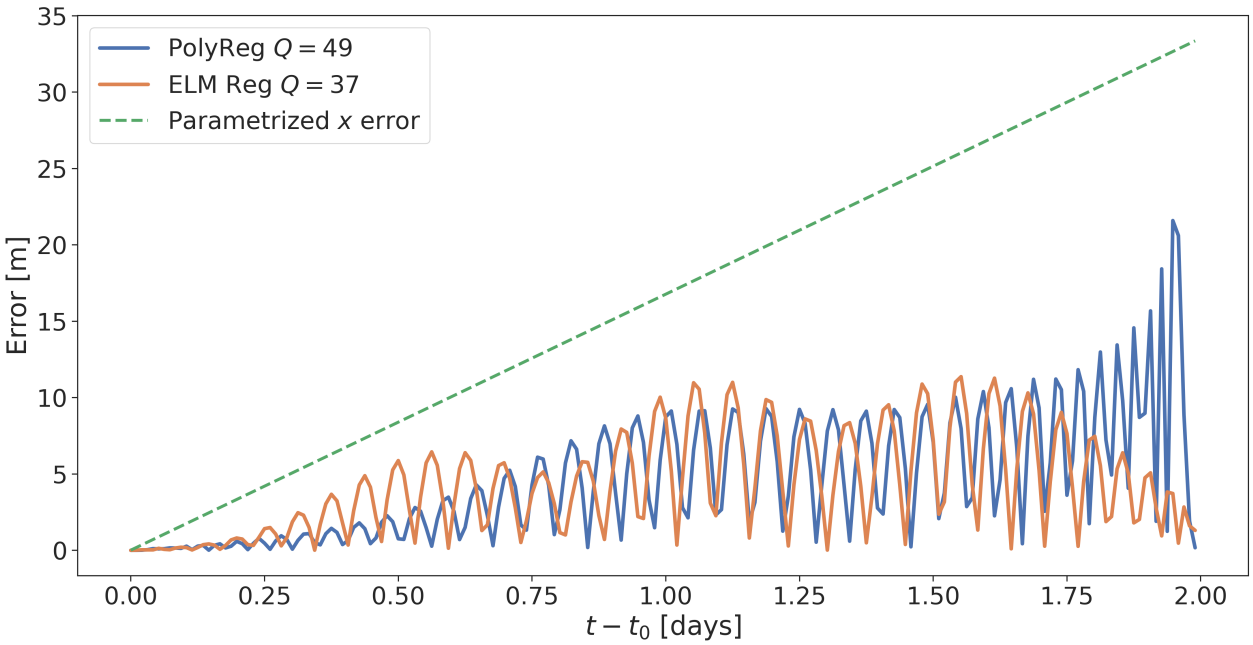

Figure 12: Weighted ELM and Polynomial regression approximation error $l(t)$ for 2 days of data.

5 Days of Data

A plot of the parametrization of the propagation error and its associated weight for 5 days of data are shown as function of time in Fig.13.

A comparison of the lowest number of coefficients required to satisfy the constraints on the approximation error between ELM and Polynomial weighted regression is shown in Table 19 and Fig. 12.

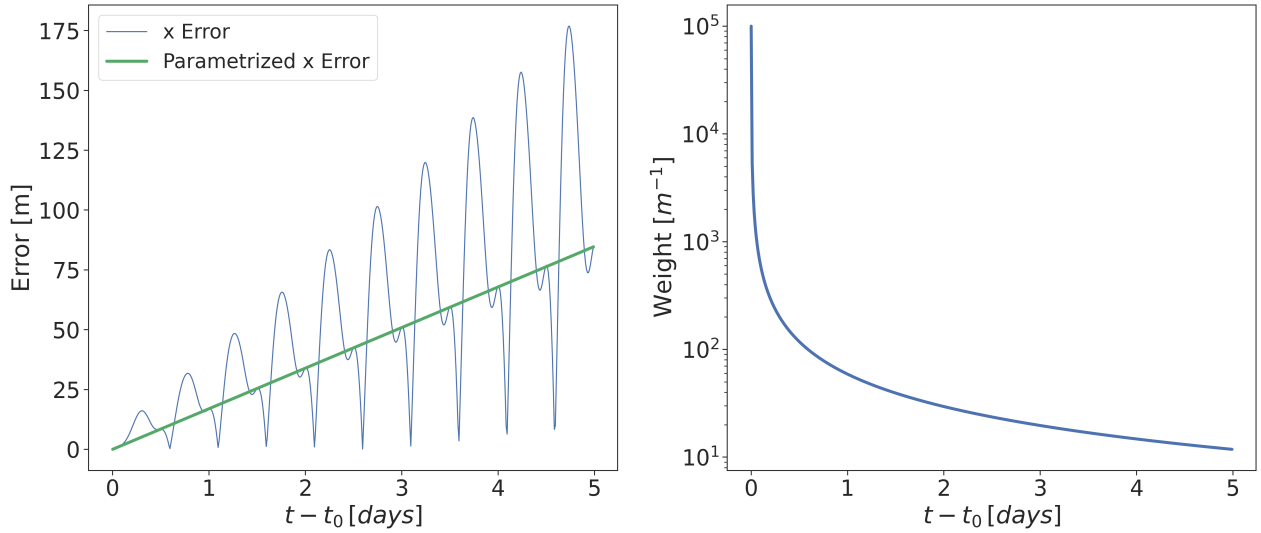

Figure 13: (**Left**)Propagation error and its linear parametrization for 5 days of data. (**Right**) Weight associated to the data points.

	Q	**CR**	**Weights Seed**	**Biases Seed**	**RMS [m]**	**Max Error [m]**
ELM	89	5.393	799907913	182516100	11.714	32.781
PolyReg	113	4.248	-	-	16.431	64.823

Table 19: Weighted ELM and Polynomial regression results for 5 days of data. $CR = 480/Q$.

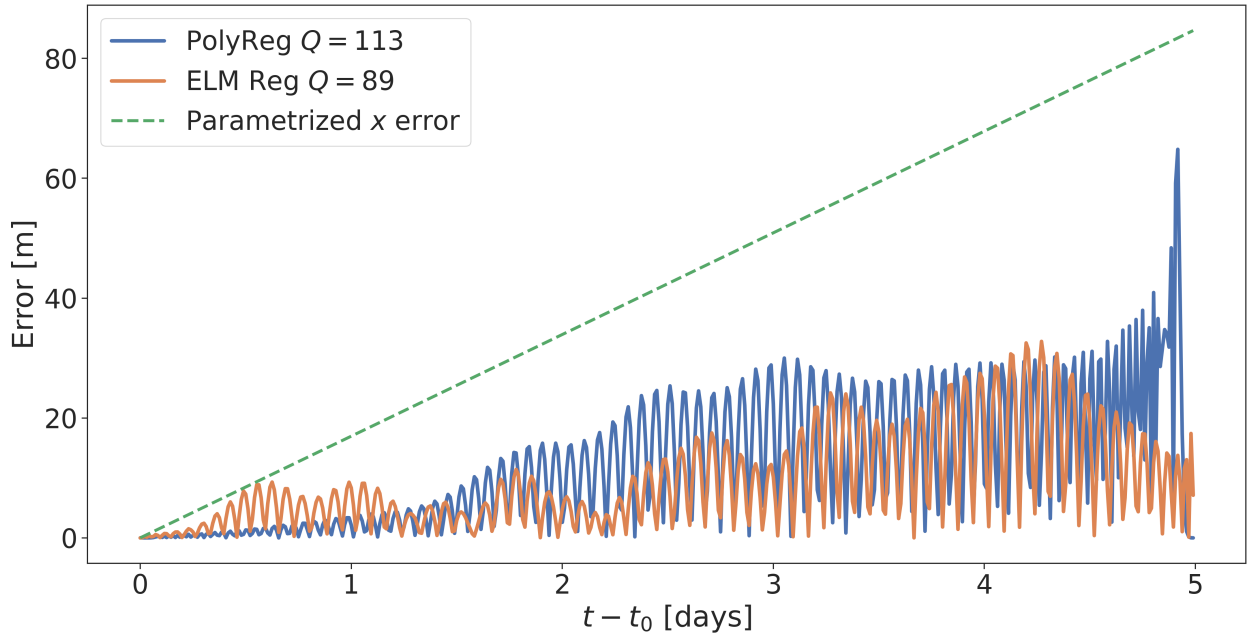

Figure 14: Weighted ELM and Polynomial regression approximation error $l(t)$ for 5 days of data.

7 Days of Data

A plot of the parametrization of the propagation error and its associated weight for 7 days of data are shown as function of time in Fig.15.

A comparison of the lowest number of coefficients required to satisfy the constraints on the approximation error between ELM and Polynomial weighted regression is shown in Table 20 and Fig. 16.

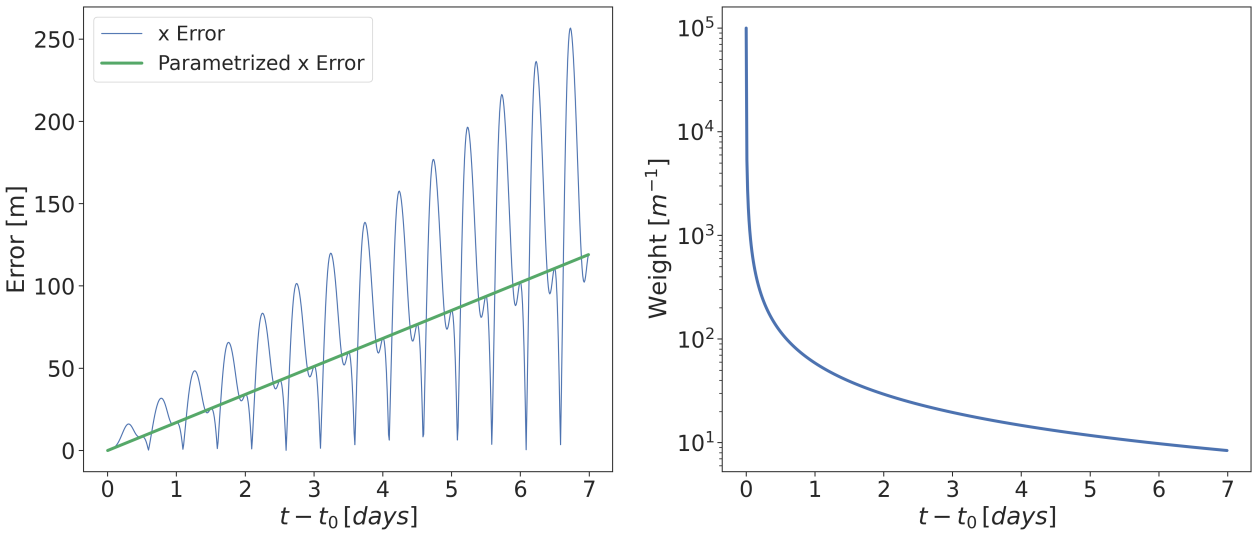

Figure 15: (**Left**) Propagation error and its linear parametrization for 7 days of data. (**Right**) Weight associated to the data points.

	Q	CR	Weights Seed	Biases Seed	RMS [m]	Max Error [m]
ELM	123	5.463	135792605	119465467	19.597	57.322
PolyReg	155	4.335	-	-	27.11	107.607

Table 20: Weighted ELM and Polynomial regression results for 7 days of data. $CR = 672/Q$.

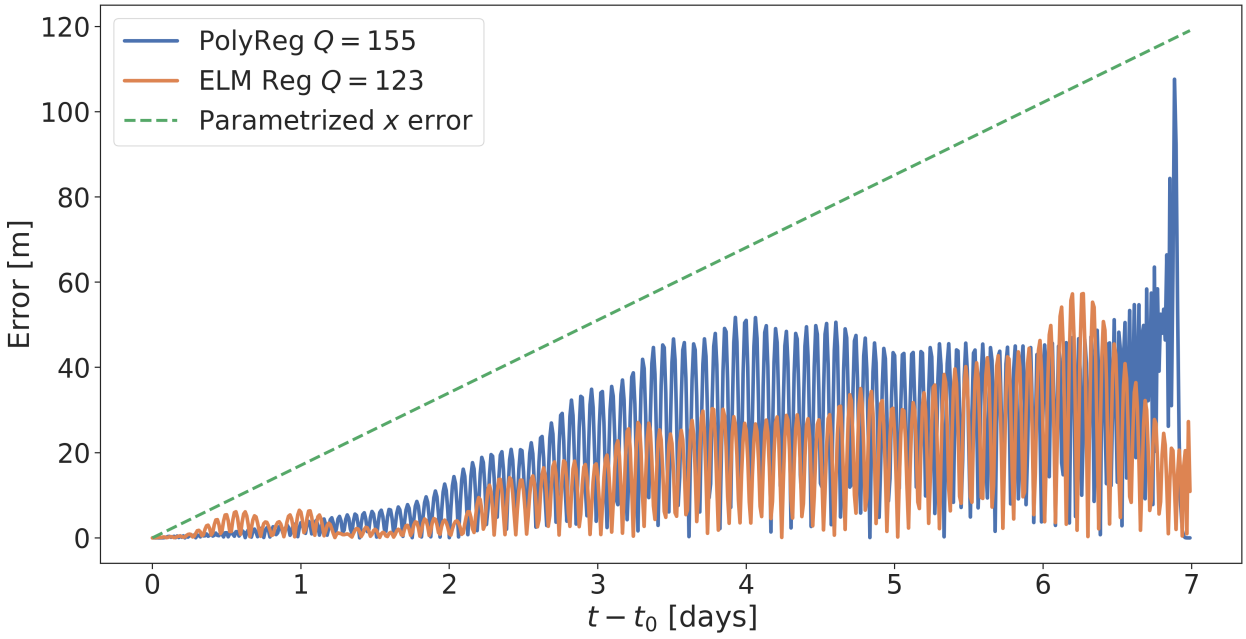

Figure 16: Weighted ELM and Polynomial regression approximation error $l(t)$ for 7 days of data.

10 Days of Data

A plot of the parametrization of the propagation error and its associated weight for 10 days of data are shown as function of time in Fig.17.

A comparison of the lowest number of coefficients required to satisfy the constraints on the approximation error between ELM and Polynomial weighted regression is shown in Table 21 and Fig. 18.

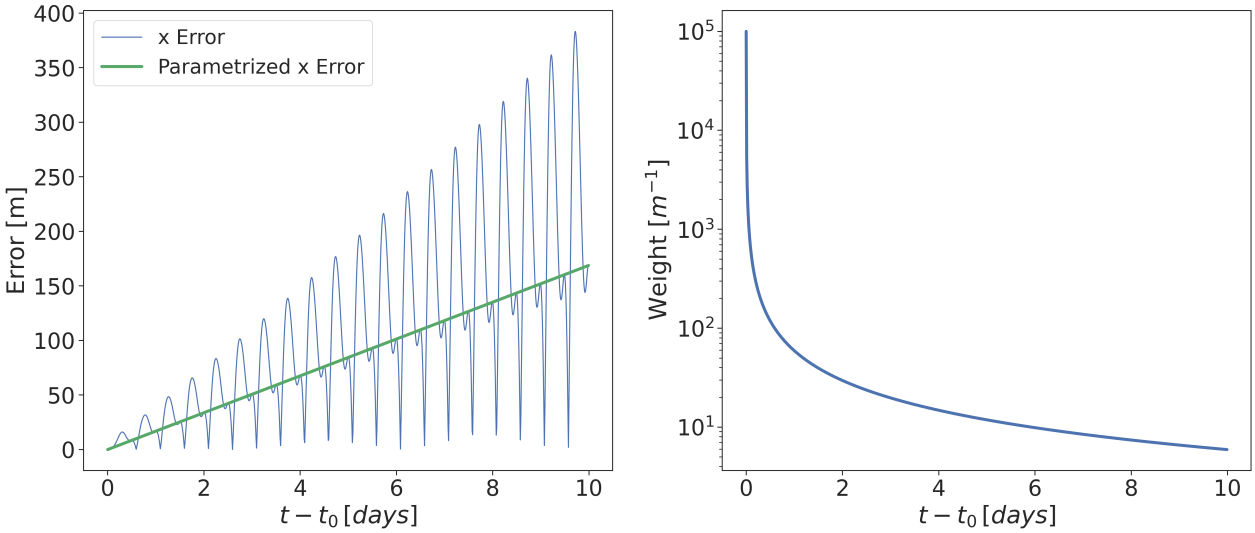

Figure 17: (**Left**) Propagation error and its linear parametrization for 10 days of data. (**Right**) Weight associated to the data points.

	Q	CR	Weights Seed	Biases Seed	RMS [m]	Max Error [m]
ELM	175	5.486	82719399	283190549	16.264	63.298
PolyReg	218	4.404	-	-	31.13	80.347

Table 21: Weighted ELM and Polynomial regression results for 10 days of data. $CR = 960/Q$.

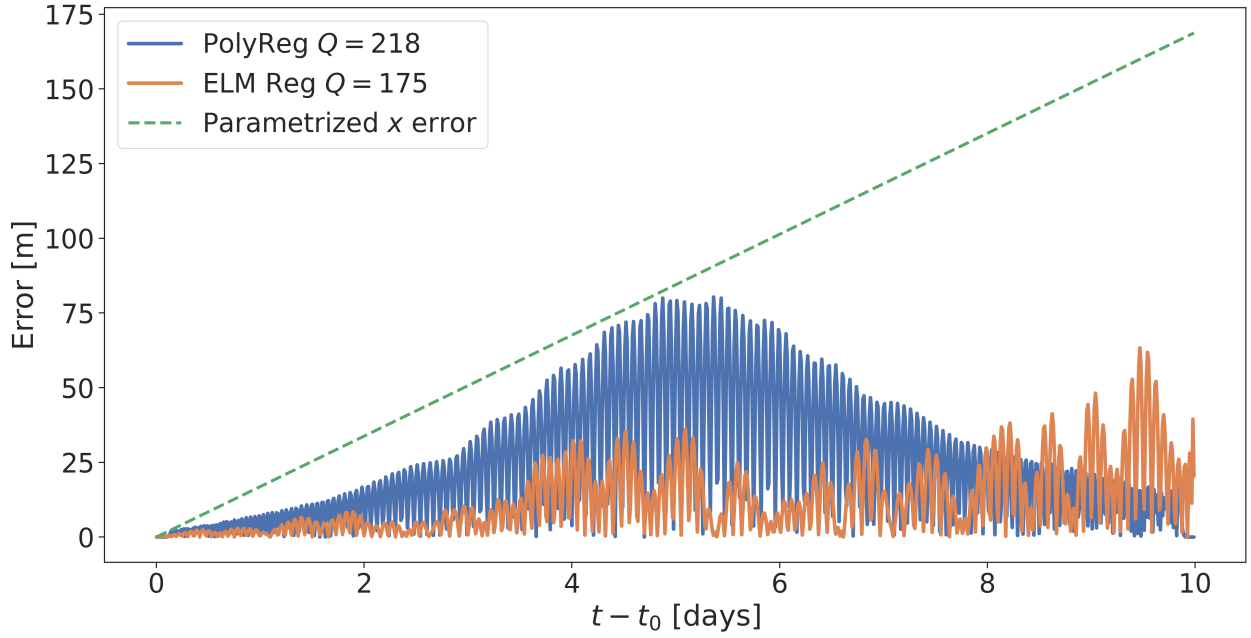

Figure 18: Weighted ELM and Polynomial regression approximation error $l(t)$ for 10 days of data.

14 Days of Data

A plot of the parametrization of the propagation error and its associated weight for 14 days of data are shown as function of time in Fig.19.

A comparison of the lowest number of coefficients required to satisfy the constraints on the approximation error between ELM and Polynomial weighted regression is shown in Table 22 and Fig. 20.

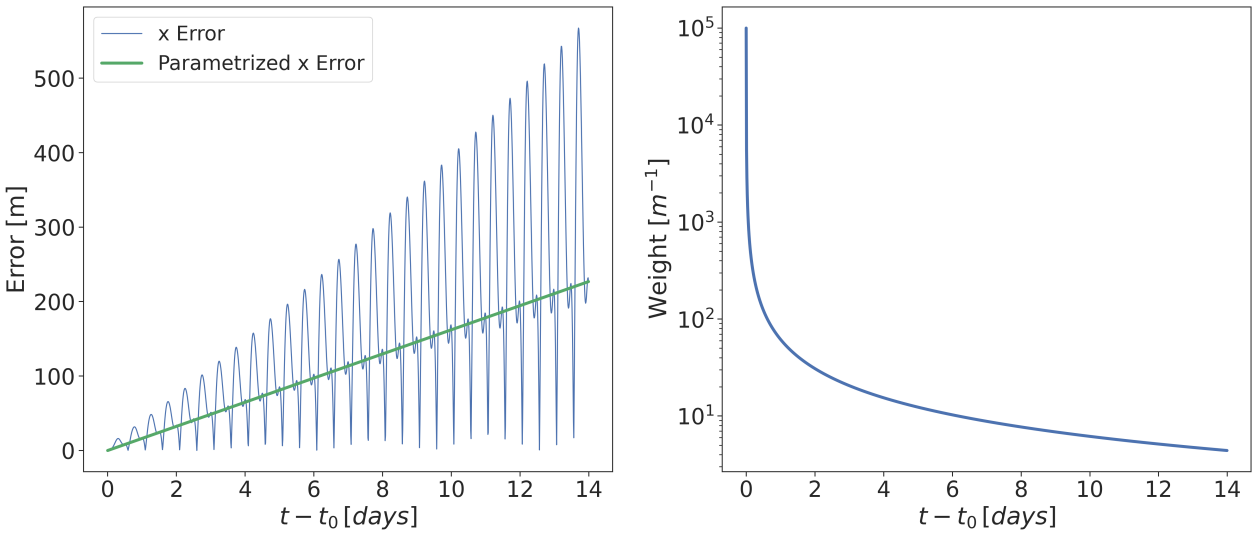

Figure 19: (**Left**) Propagation error and its linear parametrization for 14 days of data. (**Right**) Weight associated to the data points.

	Q	CR	Weights Seed	Biases Seed	RMS [m]	Max Error [m]
ELM	243	5.530	99810416	17308647	42.675	185.131
PolyReg	305	4.407	-	-	46.838	154.573

Table 22: Weighted ELM and Polynomial regression results for 14 days of data. $CR = 1344/Q$.

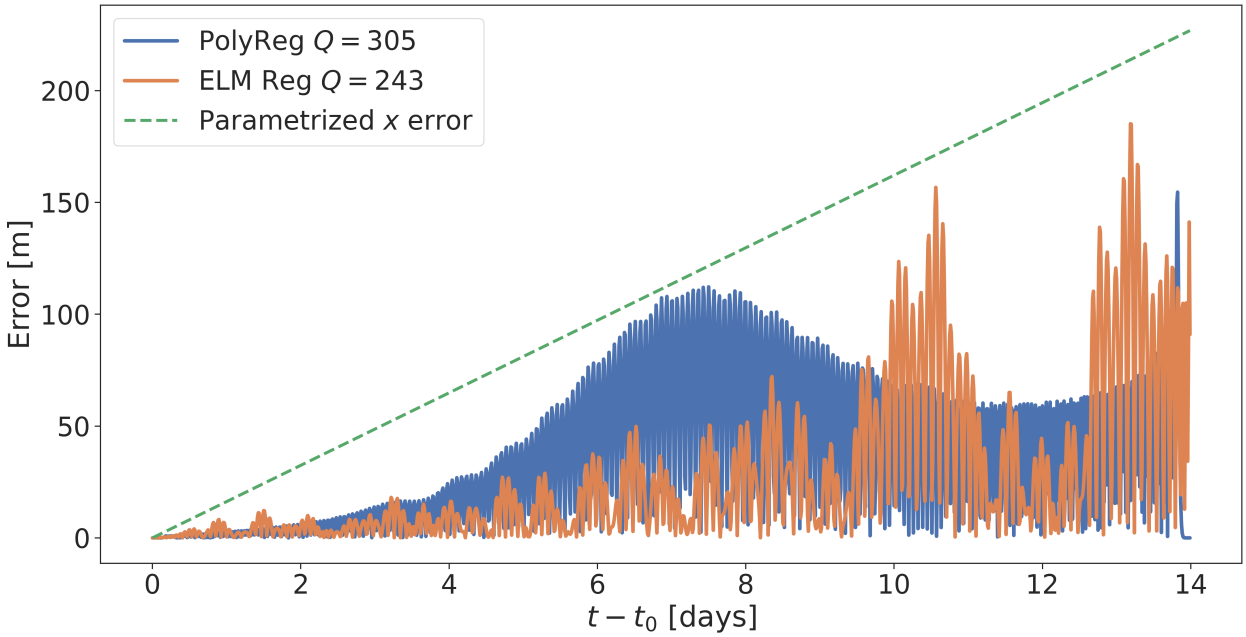

Figure 20: Weighted ELM and Polynomial regression approximation error $l(t)$ for 14 days of data.

Weighted Regression - Summary

In Table 23 and Fig. 21 the minimum number of coefficients required obtained via weighted ELM and Polynomial regression for different number of days of data are summarized.

$t_f - t_0$ (Days of Data)	$Q_{PolyReg}$	Q_{ELM}
1	28	23
2	49	37
5	113	89
7	155	123
10	218	175
14	305	243

Table 23: Summary of the minimum number of coefficients required obtained via weighted ELM nd Polynomial regression (Q_{ELM} and $Q_{PolyReg}$ respectively) for different number of days of data.

The increase of Q with respect to the number of days of data is then approximated with a power law $Q \approx (t_f - t_0)^b$ by performing a least square regression [3]:

$$Q_{PolyReg} \approx 24.48(t_f - t_0)^{0.95} \implies Q_{PolyReg} \sim \mathcal{O}\big((t_f - t_0)^{0.95}\big)$$
$$Q_{ELM} \approx 19.04(t_f - t_0)^{0.96} \implies Q_{ELM} \sim \mathcal{O}\big((t_f - t_0)^{0.96}\big)$$

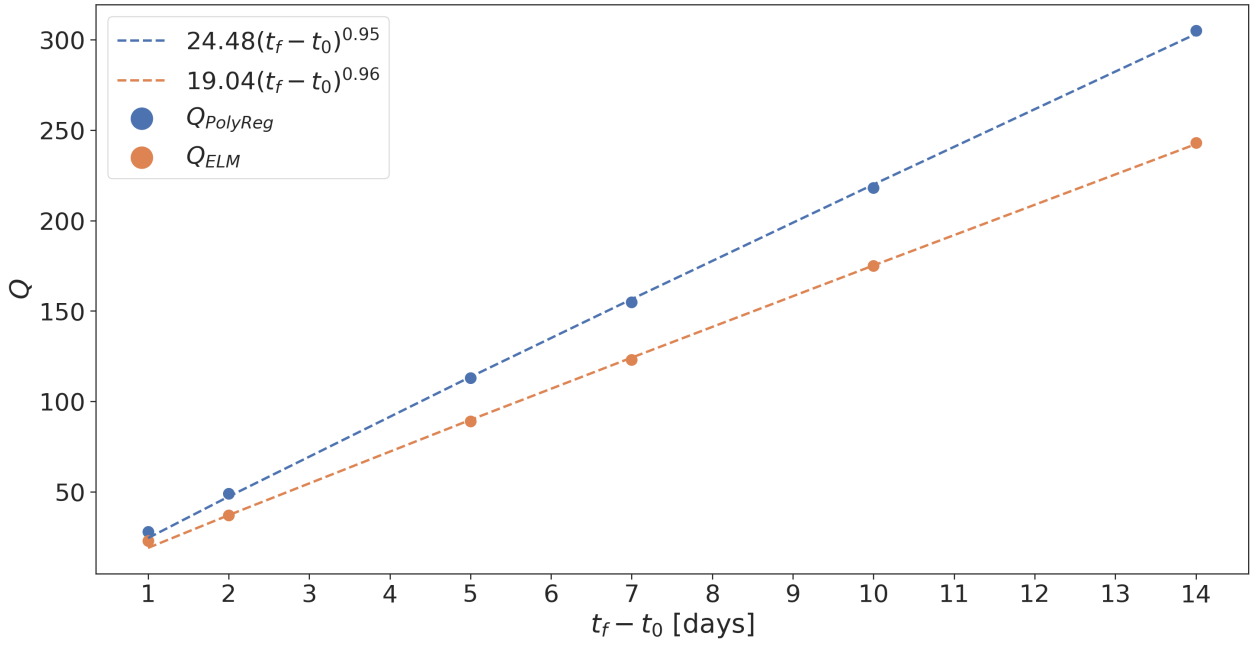

Figure 21: Minimum number of coefficients required obtained via weighted ELM and Polynomial regression (Q_{ELM} and $Q_{PolyReg}$ respectively) for different number of days of data $t_f - t_0$.
The dashed lines represent a power law increase of $Q \approx a(t_f - t_0)^b$ with the best fit coefficients a, b.
The increase of both Q_{ELM} and $Q_{PolyReg}$ is sublinear.

[3] Clearly, as for the unweighted regression, for Q_{ELM} there is some uncertainty on the power law coefficients found, given the randomness of the procedure and the few $t_f - t_0$ tried.

Part IV
Discussion

Neural Networks

First, the bad news.
According to what was written and reported in parts II-III, one might draw the conclusion that Neural Networks trained via backpropagation are just overhyped: they are difficult and time-consuming to train, require many parameters and despite this they are essentially over-complicated polynomial models.
So the question is *what is all the fuss about?*

The caveat is that in this work the scope is pretty simple: take a small number of unnoisy datapoints and fit them with the smallest number of coefficients (weights) possible. No generalization or extrapolation capabilities are required.

In fact, the main advantage of neural networks in regression problem is that they are generally more reliable in extrapolation and more robust to noisy data, thanks to the self regularization properties of optimization algorithms [17].
Furthermore, many of the successes of neural networks are obtained when they are fed with a lot of data, and such data has structure (for example in images, audio or texts), due to the rich hierarchical representations that these models are able to learn (see, for example [7], [13]).

Finally, in this work two of the main drawbacks of BackPropagation are obvious, i.e. the convergence to a local minimum of the loss in the weights space, and training time: the tried FFNN, especially the ones with sine activation function, are *in theory* perfectly capable of approximating the data well (more on this later), but the optimization algorithm (whether ADAM or SGD) hasn't been able to find the best set of weights, by getting stuck into a local minima.
Regarding training time, this is orders of magnitudes larger than ELM and Polynomial Regression ones (hours vs seconds).

For all these reasons, it is not suggested to use BackPropagation to train a Neural Network to perform ephemeris compression.

Extreme Learning Machines

A totally different discussion has to be carried out for Extreme Learning Machines.
Like traditional FFNN, they are essentially polynomials models: their output is just the linear combination of the activation functions of the hidden layer applied to the input. The correspondence between ELMs and a classical least square is made clear in section II-*Reformulation of ELM Learning as Least Squares Regression*.

From Table 9 on, one can see that in many cases, being the number of coefficients to be stored equal, ELMs with sine activation function outperform polynomial models.
This simply means that the data in question is better approximated by a linear combination of sines than by a power series.

The problem of finding frequencies ω and phases ϕ that lead to good approximation of the data is dealt with stochasticity: if there is no simple way to estimate them analytically, one just throws in some random numbers.

In practice, one tries many couples of random seeds (one for ω and one for ϕ), then for every one, generates N normally distributed combinations of frequencies ω and phases ϕ, computes analytically

the amplitudes A and finally stores the best amplitudes and random seeds which provide the best approximation of the data.

In this way, the argument of the activation functions can take any value (provided the distribution), so *by chance* it assumes the right values, which lead to a small approximation error.

Training an ELM many times, with varying random seed, will then *probably* lead to a good approximation.
Such probability will increase with the number of trials τ.

Backpropagation & ELM

Comparing the results from Tables 5 the obvious 8 question is: if ELMs are just Single Layer Feedforward Neural Networks trained in a different way, then why the results between the two learning paradigms are not even comparable? Taking the SLFNs with sine activation function trained via gradient descent, why the performances are not similar, if not better, than ELMs with sine activation function? They should be the same thing.

This divergence in results clearly showcases the problem of convergence of Gradient Descent-based BackPropagation.
In fact, as pointed out before, during backpropagation the weights might get stuck into a local minimum of the loss function, without ever exploring an important region of the weights space.
In particular, the sine activation function revealed to be very hard to train and subject to convergence issues [10].

Other activation functions (easier to train than sine) were tried, but with miserable results.

The ELM paradigm, completely solves the problems that come with BackPropagation, replacing them with stochasticity.

References

[1] BREIMAN, L. Random forests. *Machine Learning 45* (2001).

[2] CHEN, T., AND GUESTRIN, C. Xgboost. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Aug 2016).

[3] CHENG, X., KHOMTCHOUK, B., MATLOFF, N. S., AND MOHANTY, P. Polynomial regression as an alternative to neural nets. *CoRR abs/1806.06850* (2018).

[4] HEUSEL, M., RAMSAUER, H., UNTERTHINER, T., NESSLER, B., AND HOCHREITER, S. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2018.

[5] HUANG, G.-B., ZHU, Q.-Y., AND SIEW, C.-K. Extreme learning machine: a new learning scheme of feedforward neural networks. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)* (2004), vol. 2, pp. 985–990 vol.2.

[6] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization, 2017.

[7] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems 25* (01 2012).

[8] LOH, W.-Y. Classification and regression trees. *WIREs Data Mining and Knowledge Discovery 1*, 1 (2011), 14–23.

[9] ONG, H. C., CHOON, HOONG, L., TAI, , AND HUEY, S. A functional approximation comparison between neural networks and polynomial regression. *WSEAS Transactions on Mathematics 7* (06 2008).

[10] PARASCANDOLO, G., HUTTUNEN, H., AND VIRTANEN, T. Taming the waves: sine as activation function in deep neural networks.

[11] RUDER, S. An overview of gradient descent optimization algorithms. *CoRR abs/1609.04747* (2016).

[12] SERRE, D. *Matrices: Theory and application*. Springer, New York, 2002.

[13] SHERSTINSKY, A. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena 404* (Mar 2020), 132306.

[14] TSANG, W., AND MARSAGLIA, G. The ziggurat method for generating random variables. *Journal of Statistical Software 05* (01 2000).

[15] WILLIAMS, C. K. I., AND RASMUSSEN, C. E. Gaussian processes for regression. In *Proceedings of the 8th International Conference on Neural Information Processing Systems* (Cambridge, MA, USA, 1995), NIPS'95, MIT Press, p. 514–520.

[16] ZAPATA, G. The stone–weierstrass theorem and a class of banach lattice algebras. In *Aspects of Mathematics and its Applications*, J. A. Barroso, Ed., vol. 34 of *North-Holland Mathematical Library*. Elsevier, 1986, pp. 913–942.

[17] ZHANG, C., BENGIO, S., HARDT, M., RECHT, B., AND VINYALS, O. Understanding deep learning requires rethinking generalization, 2017.