Attitude propagation of Resident Space Objects with Recurrent Neural Networks

Davide Amato
Aerospace & Mechanical Engineering, The University of Arizona, Tucson, AZ, United States.
Roberto Furfaro
Systems & Industrial Engineering, The University of Arizona, Tucson, AZ, United States.
Aaron J. Rosengren
Aerospace & Mechanical Engineering, The University of Arizona, Tucson, AZ, United States.
Mohammad Maadani
Aerospace & Mechanical Engineering, The University of Arizona, Tucson, AZ, United States.

Abstract

We propose using machine learning techniques based on Recurrent Neural Networks (RNNs) for the propagation of attitude dynamics. The incorporation of a feedback loop in RNNs makes them easily adaptable for the prediction of a time series, unlike traditional, feedforward networks. A particular type of RNNs, the Long Short Term Memory networks (LSTMs), provides structures to selectively remember/forget characteristics from the past data in the time series. We develop a many-to-one RNN/LSTM architecture for the prediction of the attitude quaternion at a given time, and present training results for several choices of hyperparameters. In addition, we present a many-to-many RNN/LSTM architecture for the prediction of quaternion time series.

1 Introduction

Extensive and accurate knowledge of the rotational state of Resident Space Objects (RSOs) is crucial for several applications, but available data is scarce. For instance, attitude surveys through lightcurve inversion include only a few hundreds of objects out of the more than 17 000 currently in the Space-Track catalog [22].[1] Attitude is coupled to the orbital motion since it directly affects perturbations such as drag and solar radiation pressure, and its knowledge is pivotal for precise and long-term orbit propagation. The coupling is particularly strong for some classes of RSOs such as High Area-to-Mass Ratio objects [4]. Coupled attitude and orbit propagation is very expensive computationally, since the rotational motion induces high frequencies that require step sizes that are very small when compared to the orbital period. The problem has been tackled through a variety of approaches. Faster propagations can be achieved by partially decoupling orbital and attitude dynamics through Encke-type techniques [25] or covariance information [3]. Overall improvements can also be achieved by employing regularized perturbation methods [23], and by parallelizing implicit Runge-Kutta algorithms [10].

In our approach, we employ *deep learning* algorithms for the propagation of the attitude kinematics of a rigid body. Deep learning is an approach to Artificial Intelligence (AI) in which solutions to complex problems are learnt from observation through a hierarchy of concepts and/or computational structures. Artificial Neural Networks (ANNs) based on multiple layers fall under this paradigm. ANNs have been used in a plethora of applications for time series prediction [2], such as weather and financial forecasting [13, 15]. Most importantly, they have been capable of predicting the evolution of nonlinear dynamical systems, even exhibiting chaoticity [12]. An example in this regard is the forecasting of the number of sunspots [9]. Moreover, feedforward neural networks have been applied to orbit propagation problems, where they have been used to improve semi-analytical solutions [18].

A particular type of ANN, the Recurrent Neural Network (RNN), is able to store information on the history of the time series by incorporating connections between the hidden layer at subsequent instants. This is equivalent to the implementation of a feedback loop in the hidden layer, which is fed both the external input and the past state at each time step. RNNs perform time series prediction more efficiently than feedforward networks thanks to this characteristic, but they have been traditionally considered difficult to train due to vanishing or exploding gradients [17]. However, training of RNNs is eased by substituting the "näive" RNN layer with a bundle of recurrently connected sub-networks,

[1] `https://www.space-track.org/`, last visited August 31th, 2018.

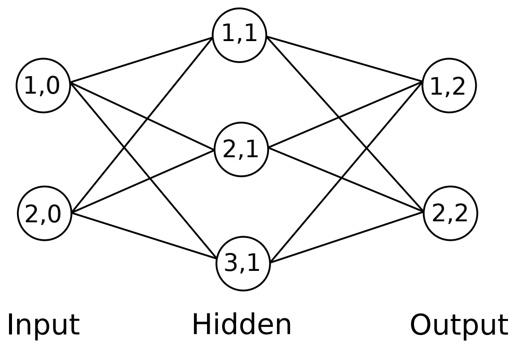


Figure 1: A feedforward neural network with input and output layers having two units each, and a single hidden layer with three units. Each unit is identified by the pair (i,l), where l is the layer number and i is the unit number within the layer. Each connection between the units has an associated weight $w_{ji}^{(l)}$.

the *Long Short-Term Memory* cell (LSTM) [11]. In an RNN/LSTM, the information can be stored over long periods of time by closing or opening sets of gates, which control the flow of information along time [7]. The LSTM cell is able to efficiently preserve the gradients along time, thereby mitigating the divergence or nullification of the gradients. RNN/LSTMs represent the current state of the art in time series processing.

This paper is concerned with the development of computational architectures based on RNN/LSTMs for the prediction of rotational motion. We present the theoretical background underlying RNN/LSTMs in section 2, and detail the implementation of two architectures for attitude prediction in section 3. The conclusions of the study, in addition to the outlook on future work, are summarized in section 4.

2 Artificial Neural Networks

An *artificial neural network* is a computing system in which the inputs are processed by *units* (or neurons). The units, each of which consists in the application of an *activation function*, are organized in successive *layers*, with the output of one layer being the input of the next one until the final (or output) layer is reached. The inputs are weighted through multiplication by constant *weight matrices*; in addition, each of the computing units also has a *bias* which shifts its activation threshold.

The structure of a neural network can be best described through graphs as that displayed in Figure 1, which depicts a simple neural network with an input layer, a single *hidden* layer, and an output layer. The example in Figure 1 is a *feedforward* network (also referred to as a *multi-layer perceptron* - MLP), as there are no connections between the units of the hidden layer: in other words, information only proceeds forward along the graph. Let $\boldsymbol{x} \in \mathbb{R}^N$ and $\boldsymbol{y} \in \mathbb{R}^M$ denote *feature* and *target* vectors respectively, which are the network inputs and outputs. The network is evaluated starting from the first layer (the input layer), whose output $\boldsymbol{o}^{(0)}$ is simply the feature vector,

$$\boldsymbol{o}^{(0)} = \boldsymbol{x}.$$

The input to unit j of the $l-$th layer, that is unit (j,l), is computed as:

$$i_j^{(l)} = \sum_{i=1}^{I_{l-1}} w_{ji}^{l} o_j^{(l-1)} + b_j^{(l)}. \tag{1}$$

The constant w_{ji}^{l} is the *weight* associated with the connection from unit i of layer $l-1$ to unit j of layer l, and the constant $b_j^{(l)}$ is a *bias*. The output of unit (j,l) is computed by appling a nonlinear activation function $\theta^{(l)}$,

$$o_j^{(l)} = \theta^{(l)}\left(i_j^{(l)}\right). \tag{2}$$

The output $o_j^{(l)}$ is passed to units in subsequent layers, until the last layer (the output layer) is reached. The output of the last layer is the target vector $\boldsymbol{y}$,

$$\boldsymbol{y} = \boldsymbol{o}^{(L)}, \tag{3}$$

where L is the number of layers. The number of hidden layers $L-1$ is the *depth* of the network.

Rewriting equations (1) and (2) in vector-matrix form eases the understanding of the network structure. We have that:

$$\boldsymbol{i}^{(l)} = W^{(l)} \cdot \boldsymbol{o}^{(l-1)} + \boldsymbol{b}^{(l)} \tag{4}$$

$$\boldsymbol{o}^{(l)} = \boldsymbol{\theta}^{(l)} \left(\boldsymbol{i}^{(l)}\right), \tag{5}$$

where $\cdot$ denotes the matrix product. For instance, a network composed by 2 hidden layers computes targets through the following system of nonlinear equations:

$$\boldsymbol{y} = \boldsymbol{o}^{(3)} \tag{6}$$

$$\boldsymbol{o}^{(3)} = \boldsymbol{\theta}^{(3)} \left(\boldsymbol{i}^{(3)}\right) = \boldsymbol{\theta}^{(3)} \left(W^{(3)} \cdot \boldsymbol{o}^{(2)} + \boldsymbol{b}^{(3)}\right) \tag{7}$$

$$\boldsymbol{o}^{(2)} = \boldsymbol{\theta}^{(2)} \left(\boldsymbol{i}^{(2)}\right) = \boldsymbol{\theta}^{(2)} \left(W^{(2)} \cdot \boldsymbol{o}^{(1)} + \boldsymbol{b}^{(2)}\right) \tag{8}$$

$$\boldsymbol{o}^{(1)} = \boldsymbol{\theta}^{(1)} \left(\boldsymbol{i}^{(1)}\right) = \boldsymbol{\theta}^{(1)} \left(W^{(1)} \cdot \boldsymbol{o}^{(0)} + \boldsymbol{b}^{(1)}\right) \tag{9}$$

$$\boldsymbol{o}^{(0)} = \boldsymbol{x}, \tag{10}$$

which is equivalent to:

$$\boldsymbol{y} = \boldsymbol{\theta}^{(3)} \left(W^{(3)} \cdot \boldsymbol{\theta}^{(2)} \left(W^{(2)} \cdot \boldsymbol{\theta}^{(1)} \left(W^{(1)} \cdot \boldsymbol{x} + \boldsymbol{b}^{(1)}\right) + \boldsymbol{b}^{(2)}\right) + \boldsymbol{b}^{(3)}\right). \tag{11}$$

Equation (11) shows that a feedforward neural network can be seen as a sequential application of vector fields. Each of the vector fields is itself obtained from the application of a nonlinear function to the sum of a matrix product and a bias.

The values of the weight matrix $W^{(l)}$ and of the bias vector $\boldsymbol{b}^{(l)}$ for each layer are learned during the process of *training*, which consists in the minimization of a loss function $\mathcal{L}$ through an optimization algorithm. The loss function is minimized over the spaces of weights and biases, and over the training samples that are provided by the user. For regression tasks, in which the neural network must predict vectors $\boldsymbol{y}_i$ of real values,[2] a common choice for the loss function is the mean square error (MSE),

$$\mathcal{L}_{\text{MSE}} = \frac{1}{m} \sum_{i=1}^{m} \|\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i\|^2. \tag{12}$$

The optimization is usually performed through a method based on gradient descent. In *mini-batch* gradient descent, the loss function is evaluated over a number of samples $m < S$, where S is the total number of samples in the training set. The samples are chosen at random until the training set is exhausted, and the process can be repeated several times to improve the convergence of $\mathcal{L}$ to a minimum. *Stochastic* gradient descent is a particular case of mini-batch gradient descent in which $m = 1$.

At each step of gradient descent, the values of each weight $w_{ji}^{(l)}$ and bias $b_j^{(l)}$ are updated iteratively by walking in the opposite direction of the gradient of the loss function:

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} \tag{13}$$

$$b_j^{(l)} \leftarrow b_j^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial b_j^{(l)}}. \tag{14}$$

The gradients of $\mathcal{L}$ with respect to the weights and biases can be computed recursively for all the layers, starting from the output layer, through the *backpropagation* algorithm [20]. The *learning rate* α is usually chosen by trial and error. The method can be made more robust with respect to local minima by adding a momentum term [19], and it has been further improved through the adoption of a variable learning rate [14].

[2]Deep learning models such as the MLP can solve a variety of tasks in addition to regression [6, section 5.1.1].

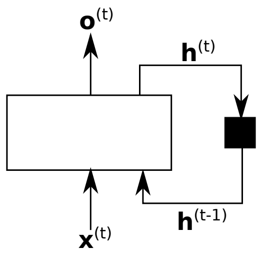


Figure 2: A recurrent neural network. The inputs to the network are the current external input $\boldsymbol{x}^{(t)}$ and the past hidden state $\boldsymbol{h}^{(t)}$. The rectangle represent the application of the activation function $\boldsymbol{\theta}$.

While MLPs perform well for pattern recognition tasks, they are not efficient for time series prediction [7]. In such an application, the features $\boldsymbol{x} \in \mathbb{R}^{T \times N}$ consist of a sequence of vectors at different time steps,

$$\boldsymbol{x}(t) \triangleq \left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right).$$

The input layer of a MLP for time series prediction has a number of units equal to the number of time steps, which makes the MLP poorly scalable for long sequences. In addition, since the weights are referenced to each time step separately, training such a network can become very expensive.

The issue can be solved by allowing connections between the hidden layer at time t and the same layer at time $t-1$, which is the principle behind the *recurrent neural network* (RNN) architecture.[3] The additional connections allow the network to store information on the previous time steps in the output of the hidden layer $\boldsymbol{h}(t)$ [6].

2.1 Recurrent Neural Networks

For purposes of illustration, we consider a recurrent neural network with a single hidden layer, following the exposition in [6]. In this context, superscripts indicate the value of the variables at any given time step t. At time $t = 0$, we specify the value of the initial input $\boldsymbol{x}^{(0)}$ and of the output state of the hidden layer $\boldsymbol{h}^{(0)}$. At following time steps, the hidden state $\boldsymbol{h}^{(t)}$ and the output of the neural network $\boldsymbol{o}^{(t)}$ are computed through the *update equations*:

$$\boldsymbol{i}^{(t)} = W \cdot \boldsymbol{h}^{(t-1)} + U \cdot \boldsymbol{x}^{(t)} + \boldsymbol{b} \tag{15}$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{\theta}\left(\boldsymbol{i}^{(t)}\right) \tag{16}$$

$$\boldsymbol{o}^{(t)} = V \cdot \boldsymbol{h}^{(t)} + \boldsymbol{c}. \tag{17}$$

The weight matrices U, V, W express the strength of the input-to-hidden, hidden-to-hidden, and hidden-to-output connections, along with the biases $\boldsymbol{b}$ and $\boldsymbol{c}$. The weights and biases are shared across different time steps, allowing the network to learn the same approximating function for the whole sequence. Sharing of weights and biases allows a RNN to generalize to longer sequences better than an MLP.

Figure 2 represents the RNN as an input and output layer, along with a hidden layer that is connected to itself. This recursive connection is delayed by one time step, since the hidden layer has access to its output from the previous step. An equivalent portrait of the neural network can be obtained by unrolling the network into subsequent time steps. In this way the flow of information along time is explicit. In practice, the RNN is always unrolled and truncated at a final time T.

Training a RNN is conceptually analogous to training a MLP. Starting from the gradient of the loss function at time T, the gradients at times $T-1, T-2, \ldots, 0$ can be computed recursively through the *backpropagation through time* (BPTT) algorithm [24]. However, RNNs suffer from a disadvantage which hinders their utilization for very long sequences: gradients of the loss function tend to either *vanish* or to *explode* along time when using BPTT or other gradient-based methods [17], suppressing the network's ability to remember long-term dependencies. Various approaches have been developed to tackle this problem, among which the most successful requires a modification of the RNN architecture by substituting the hidden layer with a sub-network denominated a *memory cell*. Such RNNs are called Long Short-Term Memory networks (RNN/LSTMs) [5, 11].

[3]This connection can also be interpretd as sharing the weights between different time steps, as highlighted in [6].

2.2 Long Short-Term Memory cell

In a RNN/LSTM, the hidden units of a standard RNN are substituted with LSTM memory cells. The LSTM cell is a sub-network operating on the current external input $\boldsymbol{x}^{(t)}$ and on the past output of the cell $\boldsymbol{y}^{(t-1)}$ through the *input, forget* and *output* gates. Information on past history is stored in the *cell state* $\boldsymbol{s}^{(t)}$. Chapter 4 of Reference [7] presents a graph illustrating the flow of information inside the LSTM cell and an exhaustive description of the algorithm.

First, the outputs $\boldsymbol{j}^{(t)}, \boldsymbol{f}^{(t)}, \boldsymbol{o}^{(t)}$ of the input, forget, and output gates (respectively) are computed through:

$$\boldsymbol{j}^{(t)} = \sigma\left(U_{\mathrm{i}} \cdot \boldsymbol{x}^{(t)} + V_{\mathrm{i}} \cdot \boldsymbol{y}^{(t-1)} + \boldsymbol{b}_{\mathrm{i}}\right) \tag{18}$$

$$\boldsymbol{f}^{(t)} = \sigma\left(U_{\mathrm{f}} \cdot \boldsymbol{x}^{(t)} + V_{\mathrm{f}} \cdot \boldsymbol{y}^{(t-1)} + \boldsymbol{b}_{\mathrm{f}}\right) \tag{19}$$

$$\boldsymbol{o}^{(t)} = \sigma\left(U_{\mathrm{o}} \cdot \boldsymbol{x}^{(t)} + V_{\mathrm{o}} \cdot \boldsymbol{y}^{(t-1)} + \boldsymbol{b}_{\mathrm{o}}\right), \tag{20}$$

where U and V are weight matrices and $\boldsymbol{b}$ are bias vectors, and each of these quantities is subscripted according to each gate. The internal state of the cell is computed according to:

$$\boldsymbol{s}^{(t)} = \boldsymbol{j}^{(t)} \tanh\left(U_{\mathrm{c}} \cdot \boldsymbol{x}^{(t)} + V_{\mathrm{c}} \cdot \boldsymbol{y}^{(t-1)} + \boldsymbol{b}_{\mathrm{c}}\right) + \boldsymbol{f}^{(t)} \boldsymbol{s}^{(t-1)}. \tag{21}$$

Finally, the cell output is computed with:

$$\boldsymbol{y}^{(t)} = \boldsymbol{o}^{(t)} \tanh\left(\boldsymbol{s}^{(t)}\right). \tag{22}$$

Although several variants on this basic architecture have been developed, none of them improves its performance radically [8].

3 RNN/LSTM architecture for attitude sequence prediction

The particular implementation of a numerical algorithm is crucial in determining its performance, and machine learning approaches make no exception to this rule. To achieve reasonable results, the theoretical description of the RNN/LSTM must be translated into an operational implementation by specifying its general architecture, which is the aim of this section.

3.1 Statement of the problem

We consider the rotational state of a rigid body to be described by an attitude quaternion $\boldsymbol{q}$,

$$\boldsymbol{q} = \{q_0; q_1, q_2, q_3\}^\top = \left\{\cos\left(\frac{\Phi}{2}\right); e_1 \sin\left(\frac{\Phi}{2}\right), e_2 \sin\left(\frac{\Phi}{2}\right), e_3 \sin\left(\frac{\Phi}{2}\right)\right\}^\top, \tag{23}$$

where e_1, e_2, e_3 are the components of the principal rotation vector and Φ is the principal rotation angle [21, chapter 3]. The evolution of the quaternion is given by the kinematic equations,

$$\dot{\boldsymbol{q}} = \frac{1}{2}\tilde{Q} \cdot \boldsymbol{\omega}, \quad \tilde{Q} = \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \end{bmatrix}, \tag{24}$$

where $\boldsymbol{\omega}$ is the angular velocity in the body-fixed frame. We specify the initial conditions for the Ordinary Differential Equation (ODE) (24) with the initial quaternion $\boldsymbol{q}(t_0)$,

$$\boldsymbol{q}(t_0) = \{0.752\,219; 0.515\,63, 0.398\,665, 0.096\,742\,5\}^\top. \tag{25}$$

The angular velocity components are prescribed as:

$$\boldsymbol{\omega} = \begin{Bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{Bmatrix} = \begin{Bmatrix} 20\sin(0.1t) \\ 0.2 \\ 20\cos(0.1t) \end{Bmatrix}. \tag{26}$$

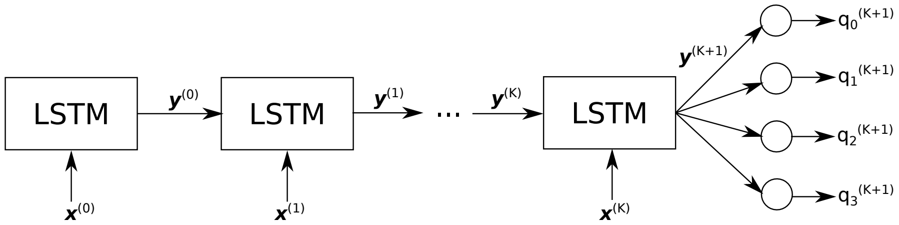


Figure 3: Unrolled RNN/LSTM architecture to predict the quaternion value at time t_{K+1}.

We compute an accurate solution by integrating Equation (24) with a 4th-order Runge-Kutta solver on K time steps of duration Δt, thus obtaining the values of the attitude quaternion $\boldsymbol{q}(t_0), \boldsymbol{q}(t_1), \ldots, \boldsymbol{q}(t_K)$. Our objective is to develop a RNN/LSTM architecture to predict the subsequent value $\boldsymbol{q}(t_{K+1})$, by training the RNN/LSTM over the initial $K+1$ values.

Note that such a RNN/LSTM will only be capable of predicting one particular solution to the kinematic equations. The generalization of the RNN/LSTM to predict any solution of the kinematic equations can be achieved by training over a large number of samples, each of which is a time series of length $K+1$ computed from specific initial conditions and angular velocity. Such a generalized RNN/LSTM might require a larger number of hidden units to provide reliable solutions, and the time required for its training will scale accordingly. However, there is no conceptual difference between RNN/LSTMs designed for the prediction of either particular or general solutions to the kinematic equations.

3.2 Architecture specification

The RNN/LSTM architecture for the prediction of the solution $\boldsymbol{q}(t_{K+1})$,

$$\boldsymbol{q}(t_{K+1}) \triangleq \left\{ q_0^{(K+1)}; q_1^{(K+1)}, q_2^{(K+1)}, q_3^{(K+1)} \right\}^\top , \tag{27}$$

is displayed in Figure 3, where the inputs (features) and outputs of the LSTM cell at time t_i are denoted with $\boldsymbol{x}^{(i)}$ and $\boldsymbol{y}^{(i)}$ respectively. The LSTM is shown as an unrolled graph for ease of visualization. The features can either consist of the value of time t_i exclusively, in which case the feature vector is 1-dimensional vector (a scalar) $\boldsymbol{x}^{(i)} = \{t_i\}$, or of the 5-dimensional vector including the current quaternion value along with time, $\boldsymbol{x}^{(i)} = \{t_i; \boldsymbol{q}(t_i)\}^\top$. We select the latter choice for the feature vector, since in this way the network can learn the relationship between subsequent elements of the time series, resulting in a model with higher capacity.[4] Also, by including t_i in the features, our aim is to learn a model that is able to generalize to different values of the time step. However, we will only consider a fixed time step for the remainder of this work.

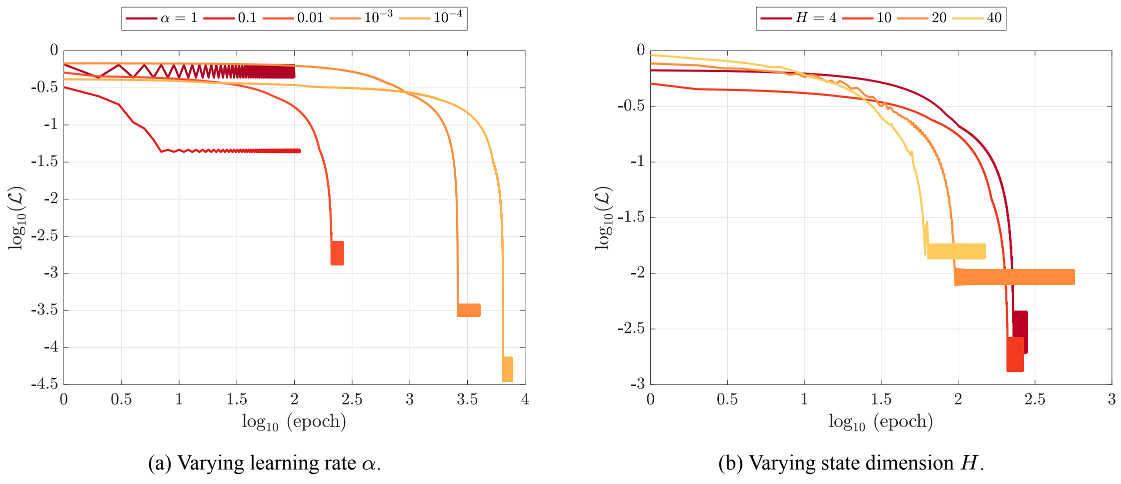
The output of the LSTM cell at the final time $\boldsymbol{y}^{(K+1)}$ is an H-dimensional vector, which is fed to a final, 4-unit layer that computes the target value of the quaternion $\boldsymbol{q}(t_{K+1})$. The activation function of the final layer is simply the identity function, i.e. each element of the quaternion is a linear combination of the outputs of the LSTM cell. Another architectural choice would be to skip the final layer and set the output of the LSTM cell to be the 4-dimensional quaternion, $\boldsymbol{y}^{(K+1)} = \boldsymbol{q}(t_{K+1})$. We avoid this approach since the number of parameters in the LSTM would be bounded by the dimensionality of the output, limiting the capacity of the model. We consider H to be a free hyperparameter to be chosen through empirical testing. The total number of parameters in the LSTM cell, i.e. the number of elements of the weight matrices and bias vectors, is $4H\,(6+H)$. Considering also the parameters of the final layer, we have a total of $4\,[H\,(7+H)+1]$ parameters to be found during training.

The loss function $\mathcal{L}$ is defined as the absolute error with respect to the true value of the quaternion $\tilde{\boldsymbol{q}}^{(K+1)}$,

$$\mathcal{L} = \left\| \boldsymbol{q}^{(K+1)} - \tilde{\boldsymbol{q}}^{(K+1)} \right\| , \tag{28}$$

where $\tilde{\boldsymbol{q}}^{(K+1)}$ is computed through an accurate numerical integration. We train the model using gradient descent on a large number of epochs. In each epoch, we perform a forward pass through the RNN/LSTM to compute the loss function. The gradient of the loss function with respect to the weights is computed using BPTT, and the weights

[4]In machine learning, the *capacity* of a model measures its capability to learn complex relationships between the features and targets.



(a) Varying learning rate α.

(b) Varying state dimension H.

Figure 4: Loss as a function of training epoch for several values of the learning rate α and of the LSTM state dimension H.

are updated through gradient descent. To mitigate the risk of exploding gradients, we clip the absolute values of the gradient components to a maximum of 10.

The RNN/LSTM is implemented using the `Keras 2.2.0` Python API for the `TensorFlow 1.8.0` library [1, 16]. All tests are performed on an 18-core iMac Pro with 2.3 GHz Intel Xeon W, with Turbo Boost up to 4.3 GHz and a 42.75 MB cache, and 128 GB of 2666 MHz DDR4 ECC memory. We do not use GPU acceleration since, given the relatively small size of the network, it does not lead to significant speed advantage.

3.3 Training and hyperparameter selection

The RNN/LSTM state dimension H, and the gradient descent learning rate α are free hyperparameters whose optimal values have to be found through numerical experiments. The search for these optimal values is made easier by the fact that different hyperparameters do not couple strongly in determining the overall RNN/LSTM performance [8]. We do not consider the training length K to be a free hyperparameter, since the LSTM cell is able to selectively remember or forget information from the past time steps in the sequence. We set $K = 150$ which, since $t_0 = 0$ and $\Delta t = 1$, implies $t^{K+1} = 150$.

The loss behavior during training for the RNN/LSTM architecture with $H = 10$ is displayed in Figure 4a for several values of the learning rate. In all cases, we performed training until the loss was observed to stabilize around a constant value. A coarse learning rate implies that once the parameters get close to the minimum of $\mathcal{L}$, gradient descent will induce large oscillations, as it can be seen for the case $\alpha = 1$. The minimum loss decreases with increasing learning rate, since the loss minimum can be pinpointed with higher accuracy. For this problem, choosing $\alpha \geq 0.01$ ensures a good performance.

The accuracy of the model cannot be improved by choosing $H > 10$, as it can be seen from Figure 4b. In fact, increasing H past a value of 10 is detrimental to the accuracy since the larger dimensionality of the parameter space makes it easier for the gradient descent algorithm to settle into a local minimum. Note that in every case considered, each training epoch takes an average of 120 ms on the machine used for the test.

3.4 Generalizing to future predictions

Since the presented architecture has only been trained on a single sample sequence, it cannot generalize to different samples. This precludes predictions for $t > t_{K+2}$, since they would require feeding to the network a sample sequence different from that on which it has been trained. In order to achieve this kind of generalization, the RNN/LSTM must be trained on the sequence $\boldsymbol{q}(t_{K+1}), \boldsymbol{q}(t_{K+2}), \ldots$ rather than on the single value $\boldsymbol{q}(t_{K+1})$. In other words, a potential RNN/LSTM architecture for attitude sequence prediction must be *many-to-many* rather than *many-to-one*. Figure 5 shows a many-to-many architecture in which the quaternion value $\boldsymbol{q}(t_{K+i+1})$ is obtained as a linear combination of the LSTM cell output $\boldsymbol{y}(t_i)$. A viable loss function for the many-to-many architecture is the root-mean-square error

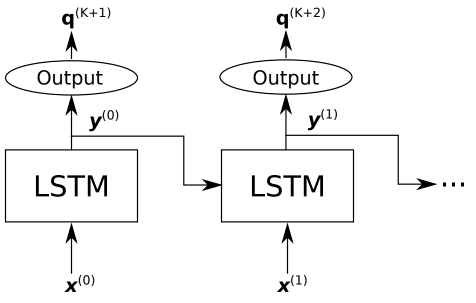


Figure 5: Many-to-many RNN/LSTM architecture for sequence prediction. The output layer has 4 units with identity activation function, as shown in Figure 3.

between the predicted and true quaternion sequences,

$$\mathcal{L} = \sqrt{\frac{1}{K}\sum_{i=1}^{K} \|\boldsymbol{q}\,(t_i) - \tilde{\boldsymbol{q}}\,(t_i)\|^2}. \tag{29}$$

The hyperparameters for the many-to-many architecture can be chosen as equal to those for the one-to-one in a first approximation, especially regarding the learning rate. The implementation and testing of the many-to-many architecture is left for further investigation.

4 Conclusions and outlook

We have devised a deep learning architecture for the prediction of attitude dynamics based on a Recurrent Neural Network with a Long Short-Term Memory cell (RNN/LSTM). We describe the rotational state of a rigid body rotating with a time-varying angular velocity through a quaternion time series. The RNN/LSTM is trained on the initial $K+1$ steps of the time series, which are computed through an accurate numerical integration, and predicts the value of the quaternion at the step $K+2$. In order to explore the hyperparameter space, we train the network for several values of the learning rate and of the dimension of the LSTM state. We find that a learning rate smaller than 0.01 and a 10-dimensional LSTM state give rise to satisfactory accuracy. In order to generalize the approach to the prediction of entire time series (similarly to what is done in numerical integration) we present a many-to-many RNN/LSTM architecture that outputs quaternion sequences, rather than a single step.

Eventually, the proposed RNN/LSTM architectures will be trained on large rotational motion data set in order to achieve a high degree of generalization. This is particularly important since RNN/LSTMs do not require any *a priori* knowledge over the system being modeled. A well-trained RNN/LSTM will thus be able to predict any kind of attitude motion by exclusively relying on previous observations, without needing knowledge of often-unknown parameters such as the mass distribution and the torques acting on a spacecraft. While we limit ourselves to a single time series in this study, the proposed architectures do not require any conceptual modification to deal with large numbers of samples. Further investigations will explore the extension of the RNNs to take arbitrary input sequence time steps, sequence lengths, and time spans.

References

[1] F. Chollet et al. *Keras: The Python Deep Learning library*. https://keras.io/. 2015.

[2] R. J. Frank, N. Davey, and S. P. Hunt. “Time Series Prediction and Neural Networks”. In: *Journal of Intelligent and Robotic Systems* 31.1-3 (2001), pp. 91–103. DOI: `10.1023/a:1012074215150`.

[3] C. Früh and M. K. Jah. “Attitude and Orbit Propagation of High Area-to-Mass Ratio (HAMR) Objects Using a Semi-Coupled Approach”. In: *The Journal of the Astronautical Sciences* 60.1 (Mar. 2013), pp. 32–50. DOI: `10.1007/s40295-014-0013-1`.

[4] C. Früh, T. M. Kelecy, and M. K. Jah. “Coupled orbit-attitude dynamics of high area-to-mass ratio (HAMR) objects: influence of solar radiation pressure, Earth’s shadow and the visibility in light curves”. In: *Celestial Mechanics and Dynamical Astronomy* 117.4 (Nov. 2013), pp. 385–404. DOI: `10.1007/s10569-013-9516-5`.

[5] F. Gers. “Long short-term memory in recurrent neural networks”. PhD thesis. Lausanne, Switzerland: Ecole Polytechnique Fédérale de Lausanne, 2001.

[6] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. Ed. by T. Dietterich. Adaptive Computation and Machine Learning. The MIT Press, 2016. ISBN: 9780262035613. URL: `https://www.deeplearningbook.org/`.

[7] A. Graves. *Supervised sequence labelling with recurrent neural networks*. Ed. by J. Kacprzyk. Vol. 385. Studies in computational intelligence. Springer-Verlag Berlin Heidelberg, 2012. DOI: `10.1007/978-3-642-24797-2`.

[8] K. Greff et al. “LSTM: a search space odyssey”. In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10 (2017), pp. 2222–2232. DOI: `10.1109/TNNLS.2016.2582924`.

[9] M. Han et al. “Prediction of Chaotic Time Series Based on the Recurrent Predictor Neural Network”. In: *IEEE Transactions on Signal Processing* 52.12 (Dec. 2004), pp. 3409–3416. DOI: `10.1109/tsp.2004.837418`.

[10] N. Hatten and R. P. Russell. “Parallel Implicit Runge-Kutta Methods Applied to Coupled Orbit/Attitude Propagation”. In: *The Journal of the Astronautical Sciences* 64.4 (Dec. 2016), pp. 333–360. DOI: `10.1007/s40295-016-0103-3`.

[11] S. Hochreiter and S. Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. DOI: `10.1162/neco.1997.9.8.1735`.

[12] H. Jaeger and H. Haas. “Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication”. In: *Science* 304.5667 (Apr. 2004), pp. 78–80. DOI: `10.1126/science.1091277`.

[13] I. Kaastra and M. Boyd. “Designing a neural network for forecasting financial and economic time series”. In: *Neurocomputing* 10.3 (Apr. 1996), pp. 215–236. DOI: `10.1016/0925-2312(95)00039-9`.

[14] D. P. Kingma and J. Lei Ba. “ADAM: a method for stochastic optimization”. In: *3rd international conference for learning representations*. San Diego, CA, United States, 2015. URL: `https://arxiv.org/abs/1412.6980`.

[15] I. Maqsood, M. R. Khan, and A. Abraham. “An ensemble of neural networks for weather forecasting”. In: *Neural Computing and Applications* 13.2 (May 2004), pp. 112–122. DOI: `10.1007/s00521-004-0413-4`.

[16] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. https://www.tensorflow.org/. Software available from tensorflow.org. 2015.

[17] R. Pascanu, T. Mikolov, and Y. Bengio. “On the difficulty of training recurrent neural networks”. In: *Proceedings of the 30th International Conference on Machine Learning*. 2013. URL: `http://proceedings.mlr.press/v28/pascanu13.pdf`.

[18] I. Pérez et al. “Application of Computational Intelligence in Order to Develop Hybrid Orbit Propagation Methods”. In: *Mathematical Problems in Engineering* 2013 (2013), pp. 1–11. DOI: `10.1155/2013/631628`.

[19] D. C. Plaut, S. J. Nowlan, and G. E. Hinton. *Experiments on learning by back propagation*. Tech. rep. CMU-CS-86-126. Carnegie-Mellon University, 1986. URL: `https://eric.ed.gov/?id=ED286930`.

[20] D. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323 (1986), pp. 533–536.

[21] H. Schaub and J. J. Junkins. *Analytical Mechanics of Space Systems*. Ed. by J. A. Schetz. AIAA, Nov. 1, 2014. 853 pp. ISBN: 1624102409. DOI: `https://doi.org/10.2514/4.867231`.

[22] J. Šilha et al. “Apparent rotation properties of space debris extracted from photometric measurements”. In: *Advances in Space Research* 61.3 (Feb. 2018), pp. 844–861. DOI: `10.1016/j.asr.2017.10.048`.

[23] H. Urrutxua and J. Peláez. “A new regularization method for fast and accurate propagation of roto-translationally coupled asteroids”. In: *AIAA/AAS Astrodynamics Specialist Conference*. 2014.

[24] R. J. Williams and D. Zipser. “Back-propagation: theory and applications”. In: ed. by Y. Chauvin and D. E. Rumelhart. Lawrence Erlbaum Publishers, 1995. Chap. Gradient-based learning algorithms for recurrent networks and their computational complexity, pp. 433–486. ISBN: 0-8058-1259-8.

[25] J. Woodburn and S. Tanygin. “Efficient numerical integration of coupled orbit and attitude trajectories using an Encke type correction algorithm”. In: *AAS/AIAA Spaceflight Mechanics Meeting*. AAS 01-428. 2001.