ASCEND
November 16-18, 2020, Virtual Event
ASCEND 2020

10.2514/6.2020-4219

Modeling and Simulation of a Spacecraft Payload Hardware Using Machine Learning Techniques

Ayush Mani Nepal * and Arnau Prat *
DLR, German Aerospace Center, 38108 Brunswick, Germany

Kilian Johann Höflinger †
DLR, German Aerospace Center, 82234 Wessling, Germany

Andreas Gerndt ‡ §
University of Bremen, 28359 Bremen, Germany
DLR, German Aerospace Center, 38108 Brunswick, Germany

Daniel Lüdtke ¶
DLR, German Aerospace Center, 38108 Brunswick, Germany

Space systems are complex and consist of multiple subsystems. Research and development teams of such complex systems are usually distributed among various institutions and space agencies. This affects the quality of the On-board Software (OBSW) since testing it without having all required subsystems at the software development site can be troublesome. In this paper, we present a data-driven method which can be used to synthesize parts of a system or even an entire system as a black-box model. We exploit the data collected from the real hardware to derive a model using a Machine Learning (ML) algorithm. The proposed model can easily be distributed among development teams and is dedicated to emulate the system for testing the OBSW.

I. Introduction

Quality assurance is an integral part of the product life-cycle of any safety critical software, such as On-board Software (OBSW) for aviation and space systems. Modern aerospace systems have a modular structure consisting of several subsystems and components [1]. These components interact with each other to fulfill a common task and ultimately achieve the mission goal. With an increasing number of components and their interactions, the complexity of the system rises [1]. Software that drives such a complex system must be robust and resilient to a wide variety of failures. In order to ensure that all features of the OBSW work as expected, they must be thoroughly tested on the component, the subsystem, and the system level.

Aerospace projects are often carried out in collaborations between agencies, research institutions, universities, or industries. In such collaborations, the development process of the system gets divided among project partners. Although an agency manufactures a hardware component, for example, it may not develop the accompanying control software. Another institute located in a different location might be responsible for the software components. In such a case, the software integration can be troublesome. In order to properly test communication modules of an OBSW, a Hardware-in-the-Loop (HiL) test environment is necessary, where all required hardware components are present.

Duplication of the missing hardware for each development site is often unfeasible due to their high complexity and manufacturing cost. When a peripheral hardware is missing, OBSW developers use mock-up methods to mimic the input/output signals of the hardware. Without deep domain knowledge, mock-up methods cannot account for the dynamic behavior of the system. Often, they produce only a random value within a given range or just return a default value. Using mock-up methods in testing the complex software may lead to incorrect test results. OBSW developers have to deal with this kind of shortcomings.

*Research Scientist, Software for Space Systems and Interactive Visualization, Lilienthalplatz 7.
†Research Scientist, Software for Space Systems and Interactive Visualization, Münchner Str. 20.
‡Professor, Center for Industrial Mathematics, Bibliothekstraße 5.
§Head of Department, Software for Space Systems and Interactive Visualization, Lilienthalplatz 7.
¶Team Leader Onboard Software Systems, Software for Space Systems and Interactive Visualization, Lilienthalplatz 7.

In this paper, we address this problem by presenting a methodology to synthesize a model with the data collected from the real payload hardware using Machine Learning (ML) techniques. The black-box model can be used to communicate with the OBSW on behalf of the real hardware to test the OBSW functionalities. We model the payload hardware along with its environment which includes modeling the behavior of sensors and actuators connected to the device over an extended period of time. We achieve this by reformulating the problem into a time series prediction problem (temporal sequence modeling), so that we can exploit well-studied ML algorithms from the field. Specifically, we explore the state-of-the-art deep learning algorithms from the Recurrent Neural Network (RNN) family.

The remainder of the paper is organized as follows: Section II gives an overview of the related work. A potential use case of this research is discussed in Sec. IV. Section III briefly presents the state-of-the-art of neural networks. Section V presents our methodology of modeling a physical system using ML algorithms. The obtained results are presented and discussed in Sec. VI. And finally in Sec. VII, a conclusion is drawn and an outlook to the future work is given.

II. Related Work

Hardware emulators have been used to perform HiL tests in various domains. In [2], a HiL simulator is developed which simulates the attitude control of a spacecraft using momentum wheels. In [3], a HiL setup is used to validate the hardware and the software of an Unmanned Aerial Vehicle (UAV). The model of the UAV is derived using the equations of motion which in case of a complex space system is usually unavailable. A neural network based diesel engine model dedicated to the HiL tests is presented in [4]. For the given displacement and the applied power, the ML model predicts the rotational speed. A similar research focusing on the gas turbine engine is presented in [5]. Both of these engine models predict only one signal at a time. In [6], a neural network based throttle actuator model is presented for the HiL simulation.

In comparison to the traditional modeling techniques, such as statistical methods, ML algorithms are popular when working with time series data [7, 8]. RNNs, such as Long Short-Term Memory (LSTM) [9] and its derivative Gated Recurrent Units (GRU) [10], are dominant ML algorithms for modeling temporal correlations from a dataset [7, 8, 11–13]. This is because of their ability to learn long-range dependencies in sequential data better than conventional RNNs. LSTMs have been used in other modeling tasks related to sensory data. In [7], multivariate time series medical data are modeled using an LSTM network to classify different diagnoses. The paper shows that an LSTM model outperforms existing baselines including the Feedforward Neural Networks (FNNs) with hand-engineered features. In [11], an LSTM is used to predict the temperature of the axle bearing of a railway. The authors trained an LSTM model with the historical temperature data along with other physical properties, such as pressure, velocity of the train, etc. The trained model predicts the temperature of the axle at six different points. We are interested in predicting more than one type of sensor values, which might be different in nature and statistical properties. In [12], an LSTM is used to model the sea surface temperature and predict future temperatures given a history of previous recordings. This univariate time series modeling approach does not account for other physical parameters.

ML algorithms have also been used in space domain applications. In [13], an LSTM model detects anomalies in the telemetry data of a spacecraft. In [14], an LSTM network is trained to predict solar flares using magnetic parameters and flare historical data. In [15], spacecraft health telemetries are processed by an unsupervised learning algorithm based on the Support Vector Machine (SVM). A SVM is used to predict coronal mass ejections from the sun in [16]. Satellite images are processed using a convolutional neural network in [17–19].

III. Background on Neural Networks

Neural networks are ML algorithms inspired by the working of neurons in biological brains for learning. The network has neurons called *units*. Each of these units has a state. These units appear in a hierarchical order; these hierarchies are referred to as *layers* in the network. Depending on the network topology, units are connected with other units of different layers via links. Each link holds a weight value which controls the information flow through them. In a neural network, combination of different types of layers are possible, such as feed-forward, recurrent, LSTM, etc. All layers in a neural network between the input and the output layer are referred to as *hidden* layers. Each of these layers takes an input vector to compute an output vector. These vectors can have multiple dimensions. For example, an LSTM layer takes a two dimensional input $[T \times P]$, where the P axis holds input values and the T axis holds the history of each input. However, while stacking different kinds of layers together, the output dimension of a layer must be equal to the input dimension of the next layer.

A. Backpropagation

Backpropagation, introduced in [20], is a successful algorithm for training neural networks [21]. In the backpropagation algorithm, the chain rule is used to calculate gradients of a cost function with respect to all weights in the network, starting from the last up to the first layer. The partial derivative gives the measure on how quickly the cost changes when the respective weight is changed. A cost function is used to compute the loss of prediction made by the network during training. A loss is a measure of deviation of the network output from the ground truth. Finally, all weights are adjusted using a scaling factor μ in the direction which minimizes the total loss. The scaling factor μ is referred to as the *learning rate*. The weights adjustment process starts at the last layer (the output layer) and is performed backwards towards the input layer. Hence, error gradients are propagating backwards in the network.

In deep neural networks, especially in RNNs, the error gradients in the front layers (the ones at the beginning of the network) become very small [22]. Thus, their contribution to the weight update becomes insignificant, hence, learning gets hindered. This phenomenon is referred to as the vanishing gradient problem [22, 23]. RNNs, such as LSTM and GRU, have been specially designed to address this problem. The other extreme of the case is the exponential increase (explosion) of the gradients. This problem occurs when the error gradients accumulate during the backpropagation and become very large [23, 24]. This results in large weight updates which destabilize the learning process. Various measures can be taken to address the exploding gradient problem, such as initializing weights with small numbers, clipping gradients to a small bound [24], etc. Nonetheless, whenever the vanishing or the exploding of gradients occurs, the neural network cannot learn.

B. Long Short-Term Memory (LSTM)

LSTMs are neural networks from the RNN family. Conventional RNNs have difficulties in learning long-ranged temporal correlations due to the vanishing/exploding gradient problem [21–23]. The LSTM is designed to overcome the shortcomings of the conventional RNN by the use of multiplicative gate units [22]. The LSTM was first introduced in 1997 by Hochreiter et al. [9] with two gate units, the *input* and the *output* gate. In 1999, Gers et al. [25] extended the LSTM architecture by introducing the third gate, namely, the *forget* gate.

The schematic of an LSTM unit with one memory cell and three gate units is depicted in Fig. 1, where i_t, f_t, and o_t are the input, forget, and output gate states at time t, respectively. And, tanh is a tangent hyperbolic function with the output range [-1, 1]. At each time step t, an LSTM unit takes two inputs (the current input x_t and the output from the previous time step h_{t-1}) and yields a single output h_t. The memory cell s_t stores the LSTM state at time t. The input gate controls the amount of input information to be added into the current memory cell. The forget gate regulates the amount of information to be removed from the memory. The output gate controls the information that flows out of the LSTM unit.

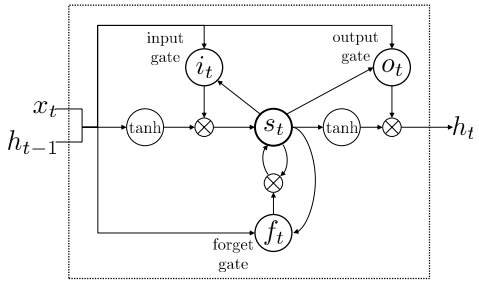


Fig. 1 Schematic of an LSTM unit with one memory cell based on [26]

IV. Use Case Space Missions

The motivation for this research comes from the first Materiewellen-Interferometrie Unter Schwerelosigkeit (MAIUS) mission [27, 28] from the year 2017. During the MAIUS-1 mission, a Bose-Einstein Condensate (BEC) was produced for the first time on board of a sounding rocket. The BEC is a special state of matter where quantum phenomena can be observed at a macro scale level. The MAIUS-1 payload was developed at the Leibniz University in Hanover. The payload hardware was used to monitor and control the experiments with the BEC. The Institute for Software Technology of the

German Aerospace Center (DLR) was responsible for developing the OBSW and drivers for the experiment control hardware [28]. The hardware was one of a kind and was not available at the DLR site. Therefore, the communication protocol of the flight software of MAIUS-1 was tested by mocking up the signals of the missing hardware.

Furthermore, future MAIUS missions as well as the NASA-DLR collaborative mission Bose-Einstein Condensate and Cold Atomic Laboratory (BECCAL) [29] aim to perform further experiments with ultra cold and condensed atoms in space. The ground software as well as the flight software being developed for these future missions are inspired by the MAIUS-1 mission [29]. Thus, the knowledge base as well as the scientific data from the MAIUS-1 mission could be exploited so that these future missions can benefit from them. Data-driven methods, like the one being presented in this paper, could be used to simulate the behavior of missing hardware components without deep domain knowledge of underlying physical processes. The MAIUS-1 mission performed complex experiments with ultra cold atoms on board of the sounding rocket. The follow-up missions are going to be even more advanced and complex. The OBSW must be able to cope with these increasing mission complexities. Using traditional mockup methods to test the functionalities of the OBSW will not be sufficient. Thus, hardware simulators for testing the OBSW will become more crucial for future space missions.

V. Methodology

In this section, we present our methodology to synthesize a model using data collected from the real hardware. We model the hardware system along with the actuators and sensors connected to it. In order to achieve that, we reformulate the task of modeling multiple physical processes into a multivariate time series prediction problem and solve it using the state-of-the-art algorithm from the field of temporal sequence modeling. In such a formulation, an actuator acts as a cause and controls the state of the system. Thus, it is an independent variable because its state is changed from outside of the system, for example, through commands sent by the OBSW. On the other hand, sensors observe the system and its environment, thus their value is dependent to the state of the system. From the OBSW perspective, every write command to an actuator is an input to the system and every read command to a sensor is an output from the system. During training, the ML algorithm learns to map these input variables to target variables over time. The trained model predicts sensor readings for a given set of actuator states and their history.

The methodology of our approach consists of four main steps. These steps are executed sequentially as depicted in Fig. 2. In the first step, we design and perform a laboratory experiment with the real hardware. We record network data packets exchanged between the hardware and the OBSW during the experiment. From the recorded packets, we extract the actuator and sensor values and prepare the training dataset in the second step. In the third step, we design an LSTM model and train it using the data prepared earlier. Finally, we evaluate the performance of the trained model.

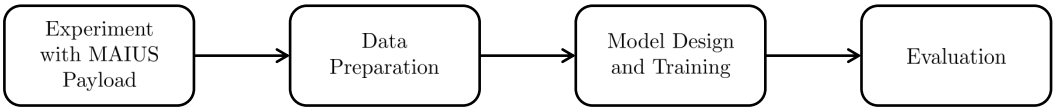


Fig. 2 Four main steps of the methodology

A. Experiment with the MAIUS Payload

We design a laboratory experiment using a part of the MAIUS payload hardware. The payload hardware used in the MAIUS-I mission consists of several stacks of printed circuit boards (PCBs) [28]. Each of these PCBs is referred to as a *card*. Every card is designed for a specific purpose and provides input and output channels where external devices, such as actuators and sensors can be connected. For our laboratory experiment, we use a stack with seven cards.

The experiment is designed with the components listed in Table 1. We choose these components so that various kinds of signals such as binary, step, continuous, etc. are incorporated in the experiment. Since our hypothesis takes the system as a black-box, the modeling technique does not require information on how these components are connected with the payload hardware. Nevertheless, we connect these components in such a way that inter-dependencies between them are created and different physical processes are defined. Through our modeling approach, we intend to simulate these physical processes.

Table 1 List of sensors and actuators used in the experiment

# Component	Signal Type
1 temperature sensor	continuous
2 photodiodes	step
1 rotatory encoder	constant
1 resistor	binary
1 fan	binary
3 LEDs	step

The experimental setup is depicted in Fig. 3. Everything inside the dotted box is modeled by a single LSTM network. A fan, a resistor, and a temperature sensor are kept close to each other, so that the temperature reading is dependent on the state of the fan and the resistor. The fan and resistor only have two states, they can be turned on and off. Thus, they make up two independent binary variables. Depending on their states, the temperature sensor produces continuous values. Two LEDs, led-1 and led-2, are connected together; both of them are faced towards photodiode-1. In the same way, led-3 is faced towards photodiode-2. Depending on the provided voltage level, the brightness of the LEDs differs, which is reflected in the photodiode readings. The LEDs and photodiodes together make up step signals. There is a divider between the photodiodes. The divider is intended to block light from the LEDs. All LEDs and both photodiodes are mounted inside a box. A rotatory encoder is fixed to value 45 and never changed, thus it is constant throughout the experiment. Each of these components is connected to a channel of the MAIUS hardware. The ground computer sends commands to the OBSW to set actuator states and to get sensor readings. On the other end, the OBSW delegates respective commands to the MAIUS hardware. We use a script in the ground computer to dispatch command messages to the On-board Computer (OBC) automatically.

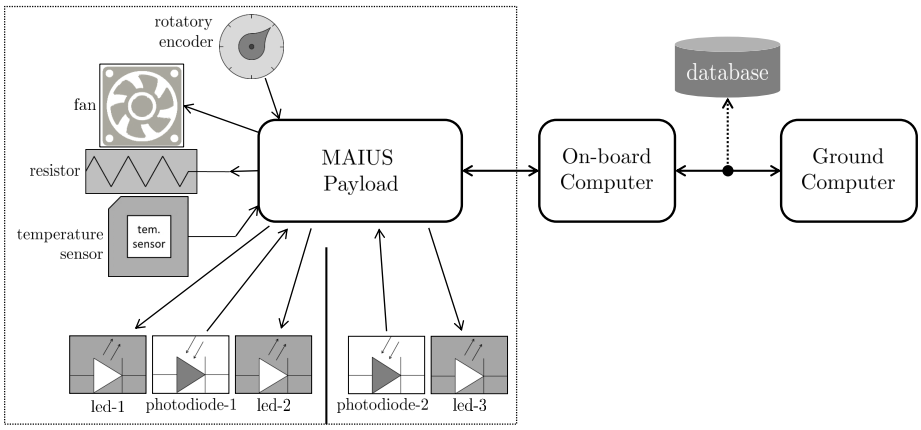


Fig. 3 Setup for the experiment with the MAIUS payload hardware

During the experiment, the fan and the resistor are turned on and off in a random interval between 2 to 20 seconds and the temperature is monitored every 500 milliseconds. Similarly, the voltage across all three LEDs is changed randomly to 0, 2.5, and 5 Volts, also while making random pauses of 2 to 20 seconds. During the pause, all LEDs keep their assigned voltage level. Similar to that of the temperature sensor, photodiode readings are also taken in the same interval. The experiment is conducted for about 90 minutes. During the experiment, every message on the communication line is recorded, and finally when the experiment is over, they are saved to the file system. This way of data acquisition, i.e., recording data from the communication line and not from the software, does not require any changes to the OBSW and it also does not interfere with the normal OBC operations.

B. Training Data Preparation

The recordings from the experiment are filtered to extract the input/output channel values. The channel values are collected as time series data. The sampling rate of the time series is equal to the data acquisition rate, i.e., 500 milliseconds. All output signals are truncated to the length of the shortest signal and then concatenated together to form an $[N \times Q]$ matrix, where N is the number of data points in time (samples) and Q is the length of the target vector (number of output signals). In our laboratory experiment, there are $Q = 4$ output signals (two from the photodiodes, one from the rotatory encoder, and one from the temperature sensor). For each sample in the output matrix, an input sample is constructed from the input time series data. At every point in time where the input value is missing, the value from the previous time step is used, i.e., the actuator remains in the same state. Hence, the input matrix has the shape $[N \times P]$, where P is the length of the input vector. The experiment has five input signals, thus $P = 5$.

At the next step, both matrices are normalized to have zero mean and unit variance. This scaling is necessary to ensure that all input and target values are in a comparable range. If the data values are not scaled properly, backpropagation emphasizes one feature/attribute of the data over the other which results in equally important features contributing differently to the weight update, resulting in a higher error rate [30].

The input of an LSTM model has the shape $[T \times P]$, where T is the sequence length. Since each time step corresponds to a layer in an RNN [24], using all N values as one long sequence and feeding the whole input matrix as a single sample to the LSTM is computationally expensive and increases the risk of numerical instabilities in the network [23, 24]. Therefore, the input matrix is transformed so that each sample has a sequence length of T, where $T \ll N$. We use the sliding window technique to transform the input matrix to have a shape of $[N \times T \times P]$. This is done by appending $T - 1$ preceding samples to each input sample. After this transformation, N gets shortened by $T - 1$ samples, because the first $T - 1$ samples are used as the history of the first sample. In order to maintain the one-to-one mapping between the input and target matrices, first $T - 1$ samples are removed from the target matrix. We tried different sequence lengths ranging from 100 up to 600; the sequence length of $T = 400$ performed well for our dataset.

Finally, both datasets are divided into three parts, namely, the training set, the validation set, and the test set. The training set contains 70% of the total samples. Other 20% of the samples are used to validate the model during training. And the remaining 10% are reserved as the test set. The test set is never used during training, but only used to evaluate the model's performance once the training is complete. This division of the dataset is done while maintaining the alignment between the input and target counterparts.

C. Model Design and Training

In this section, we present our LSTM model architecture and explain the training procedure. The training data are collected from various hardware components. Different hardware components produce different types of signals. These signals can vary heavily in their statistical properties. The model must be resilient to such a highly diverse dataset. This is achieved by appropriately choosing the value of hyperparameters of the model. Hyperparameters are those parameters of a neural network which are not tuned automatically during training but are determined before the training begins. A neural network is configured by tuning its hyperparameters so that it works better with the given dataset. There are many tunable hyperparameters in a neural network; adjusting them manually is tedious. It is considered one of the most difficult tasks in deep learning [31]. We use the Bayesian optimization algorithm based on the Gaussian process described in [32] to automate the process of finding optimal values for the hyperparameters of our model.

1. Design

The structure of our LSTM model is illustrated in Fig. 4, where TD-Dense represents a time-distributed dense layer and BN is a batch normalization layer [33]. The number of units in each layer is given in brackets. Output dimensions are shown at the transitions, where n represents the number of samples. Usually, n is much smaller than the total number of available training samples N.

The model is categorized into three main parts: the memory block, the parallel blocks, and the output. Empirically, we saw that neural networks have difficulties in correctly producing all Q signals together through a single sequential structure. Our targets are the temperature sensor, the photodiodes, and the rotatory encoder (see Table 1). The signals produced by them are not comparable to each other in their nature. Therefore, each of them requires a different set of hyperparameters which is not possible if there is a single sequential structure. This problem rarely occurs in other neural network applications, as they usually only deal with similar kind of targets, such as images, speech data, or categories of classification.

We address this problem by splitting the network into Q parallel *neural branches*, i.e., one for each target. This

allows us to predict various kinds of signals at once without having to compromise on the prediction accuracy. Every neural branch contains only a fraction of weights than the large single sequential structure. All neural branches still share the memory block, which is the one with the highest number of weights, so the majority of weights in the network are still shared. Moreover, each branch can have an individual set of hyperparameters, so it can be configured differently to produce a specific signal.

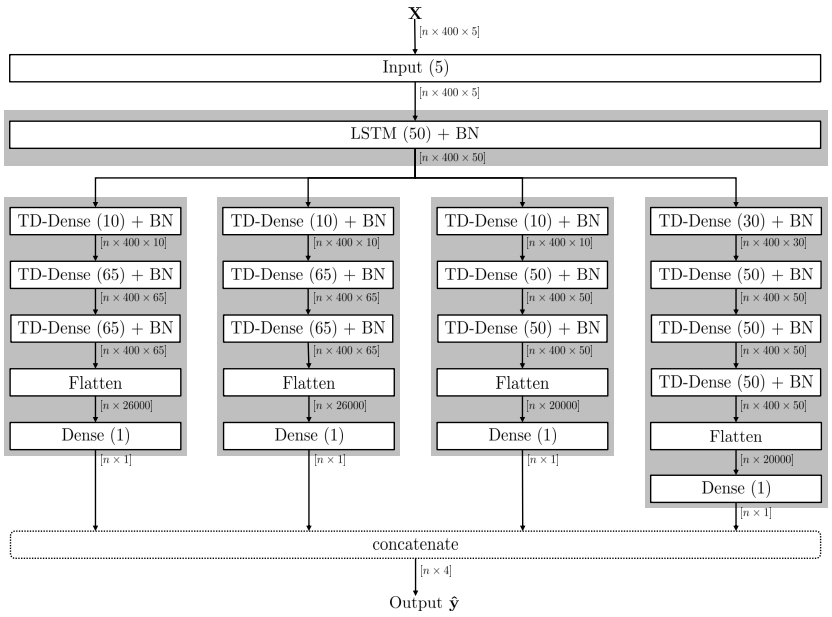


Fig. 4 Model architecture showing a common memory block and parallel neural branches

The memory block consists of only one LSTM layer with 50 units followed by a Batch Normalization (BN) layer. The LSTM layer serves as a memory of the model and is configured to return the full sequence (all 400 time steps) and not only the last sequence. Hence, for each input sample of shape $[T \times P]$, the LSTM layer outputs $[T \times 50]$ values. While training deep neural networks, the distribution of each layer's input changes, as the weights of the previous layers change; this phenomenon is referred to as *internal covariate shift* [33]. This effect slows down the training significantly and demands lower learning rates and carefully engineered weight initialization techniques [33]. The BN layer addresses this problem by allowing normalization to be a part of the model architecture and performs normalization on each *mini-batch* during training. A mini-batch is referred to the number of input samples the network sees before averaging the gradients to update weights.

Each of the neural branches is made up of multiple time-distributed dense layers, each of them followed by a BN layer. Dense layers are feed-forward layers, which are fully connected with the preceding as well as the succeeding layer. All of these fully connected layers are time distributed, which means the same layer is shared across all 400 time steps. At the tail of each neural branch, a Flatten layer is connected whose sole purpose is to reduce the dimension and possesses no trainable weights. Finally, the output of the flatten layer is fed into the output layer. The output layer is a feed-forward layer with only one unit, hence, each neural branch predicts one time step of the output signal for a given input sample. It is interesting to notice that the first layer of each neural branch has significantly fewer units than the succeeding layers in the branch. We found that this bottleneck serves as a filter and helps to discard irrelevant activations from the memory block and allows only the necessary ones to pass into the neural branch.

Furthermore, we introduce non-linearity into the model by using different activation functions. An activation function is applied to the output of the LSTM layer and each of the time-distributed dense layers. The tanh function is used in the forward links of the LSTM layer and the Sigmoid function $\sigma \in [0, 1]$ is used in the recurrent links. These are the default activation functions used in the LSTM implementation in popular ML frameworks. The output of each time-distributed dense layer is applied with the Rectified Linear Unit (ReLu) [34, 35] function. ReLu is a half-wave rectifier $f(h_j) = \max(0, h_j)$, where h_j is the output of jth unit of an arbitrary layer. The ReLu is a popular activation function because it usually accelerates the learning process in deep neural networks [36]. Using activation functions

after the BN layer would defeat its purpose and since the Flatten layer does not perform any numerical computation, no activation function is used after those layers. Moreover, in regression, the basic idea is to manipulate the input by the hidden layers in a non-linear manner so that a linear discrimination is possible in the output layer [36], therefore, a linear activation function $f(h_j) = h_j$ is used in the last layer of each neural branch.

2. Training

Adam, a first-order gradient-based stochastic optimization algorithm introduced in [37], is used as the learning method. We configure Adam to clip the Euclidean norm (L_2) of all gradient vectors to 1.0, which means, if the vector norm of a gradient is more than 1.0, then all values in the vector are re-scaled so that its L_2 norm becomes 1.0. In addition to that, we also clip the absolute value of gradients between -1.0 and 1.0. These are techniques to tackle the exploding gradient problem in deep learning. These clipping values were determined experimentally. As a cost function, the Mean Squared Error (MSE) is used. MSE is one of the popular measures for quantifying regression models. It is easy to understand and also easy to compute. Before an optimal training session begins, many other hyperparameters are optimized. They are discussed briefly below:

1) **Mini-batch Size**: By convention, all samples in the training data are collectively referred to as a *batch for training*. However, during training, input samples from the training data are fed into the network in smaller chunks referred to as mini-batches. In the backpropagation algorithm, error gradients are averaged over a mini-batch of samples. Therefore, if the mini-batch size is too small, weights are updated more frequently, which increases the risk of diversion from their optimum value. On the other hand, if the mini-batch size is too large, the learning method averages gradients over a large number of samples, which results in the overfitting of the model to the training set. Overfitting is a condition when the model learns unnecessary details from the training set and performs well with it but fails to predict correctly when unseen inputs are given. Overfitting decreases the generalization capability of the model. Hence, mini-batch size is an important hyperparameter and must be optimized. We determined an appropriate mini-batch size for our model to be 256 samples.
2) **Number of Epochs**: An epoch of training is said to be finished when the entire training data has been passed through the network once. Usually, a neural network is trained for several epochs before its accuracy converges to the optimum. Training few numbers of epochs may not be enough for the model to learn sufficiently. On the other hand, training for many epochs may result in the overfitting of the model. Number of epochs is a hyperparameter and must be optimized. We found that training for just 10 epochs is sufficient for the LSTM model to produce good results for the given task.
3) **Initial Learning Rate**: While training using the stochastic gradient descent method, the learning rate is the size of the step taken during the descent towards the minimum of the cost function. Using larger steps may overshoot the global minimal point whereas too small values increase the risk of convergence to a local minimum. Therefore, the learning rate is a very important hyperparameter while training a neural network. Ideally, one begins with a relatively large initial learning rate to approach the minimum faster and then decreases it during training with an attempt to perfectly land on the global minimum. In this study, we used the callback API from the Keras ML framework to develop a learning rate scheduler which monitors the training process and adjusts the learning rate if the validation loss did not sufficiently decrease for three consecutive epochs. The developed scheduler also has a fallback procedure which restores the last set of best weights in the network when the training performance starts to degrade at the end of an epoch. The algorithm uses MSE on the validation set for quantifying the model performance during training. We determined an appropriate initial learning rate of 0.002 and an initial decay rate for the scheduler of 0.01.
4) **Weight Regularization**: Regularization is a way to penalize large weights so that they do not become too dominant. Large weights emphasize some features which may or may not be relevant. To avoid irrelevant details from the training set overshadowing important features, it is essential to have all weight values to be in a comparable range. Weight regularization is also an effective technique to address the overfitting problem in neural networks. A large regularization factor results in a harsh penalty causing the loss of important feature information. On the other hand, too small values might be ineffective or might be an insufficient penalty allowing larger weights to remain dominant. Thus, the regularization factor is an important hyperparameter and has to be chosen for each layer in the network with trainable weights. In our model, all feed-forward layers have a weight regularization factor of 0.4 with an exception to the dense layers of the neural branch producing the continuous signal; there a higher regularization of 0.9 is required. The weights in recurrent links of the LSTM layer also required a regularization of 0.9.

5) **Weight Initialization**: In stochastic learning methods, weights in the network are initialized randomly and then optimized during training by the backpropagation algorithm. Initial weights are the starting points from where the learning begins. Therefore, how weights are initialized in a neural network influences the stability, efficiency, and accuracy of the learning process. For instance, initializing weights with large values increases the risk of exploding gradients during the backpropagation and may also cause other instabilities, such as numerical overflows. Ideally, initial weights are drawn from a normal or an uniform distribution with a low variance. We initialize weights from the *glorot uniform* [38] distribution for the forward links in the LSTM layer. Improved performance is observed when the recurrent links of the LSTM layer and all dense layers in the network are initialized from a random orthogonal distribution.

D. Evaluation Measures

In order to evaluate the prediction capability of our model, 2000 samples in the test set are used. These samples were never seen by the network during training. Since values in the test set are scaled as mentioned in Sec. V.B, predicted values are also in a scaled form. After all predictions have been made, the predicted data are descaled to their physical representation. The prediction accuracy of the model is quantified against the ground truth by computing two scale-free metrics, namely, the Mean Absolute Percent Error (MAPE) and the symmetric MAPE (sMAPE). These metrics are standard in practice for the performance evaluation of time series prediction. Formulae to compute these metrics are given in Eq. (1), where N_{test} is the total number of samples in the test set, y_n is the true value of the n^{th} sample, and $\hat{y}_n$ is the predicted value.

$$\text{MAPE} = \frac{100}{N_{test}} \sum_{n=0}^{N_{test}} \frac{|y_n - \hat{y}_n|}{|y_n|}, \tag{1a}$$

$$\text{sMAPE} = \frac{200}{N_{test}} \sum_{n=0}^{N_{test}} \frac{|y_n - \hat{y}_n|}{|y_n| + |\hat{y}_n|}. \tag{1b}$$

In order to manually test the developed LSTM model, we use it to drive a simulator application. A screenshot is shown in Fig. 5. The simulator is a simple Python Graphical User Interface (GUI) application where actuator states can be changed by the user at any time. The control buttons seen in the left panel can be used to toggle the resistor and the fan. The spin buttons can be used to change the voltage level of the LEDs. The simulator periodically makes predictions using the trained LSTM model in the background. When an input is changed by the user, the change is used by the LSTM immediately. The predicted sensor values are displayed in the live plot. We use the simulator application to manually analyze the trained LSTM model. The application can be extended to communicate with the OBSW. The picture of the hardware seen in the left-bottom corner of the GUI is a small prototype of the experiment control hardware from the MAIUS-1 mission.

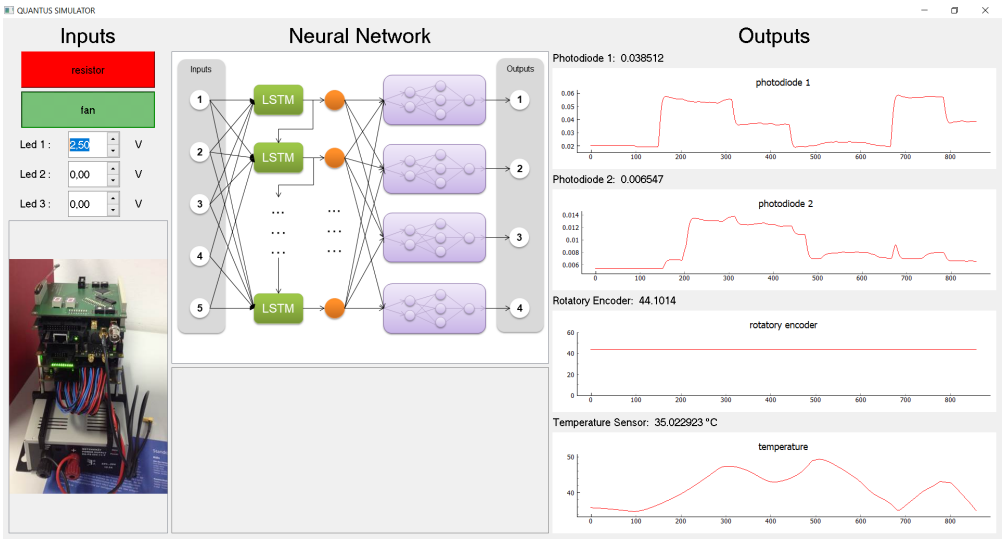
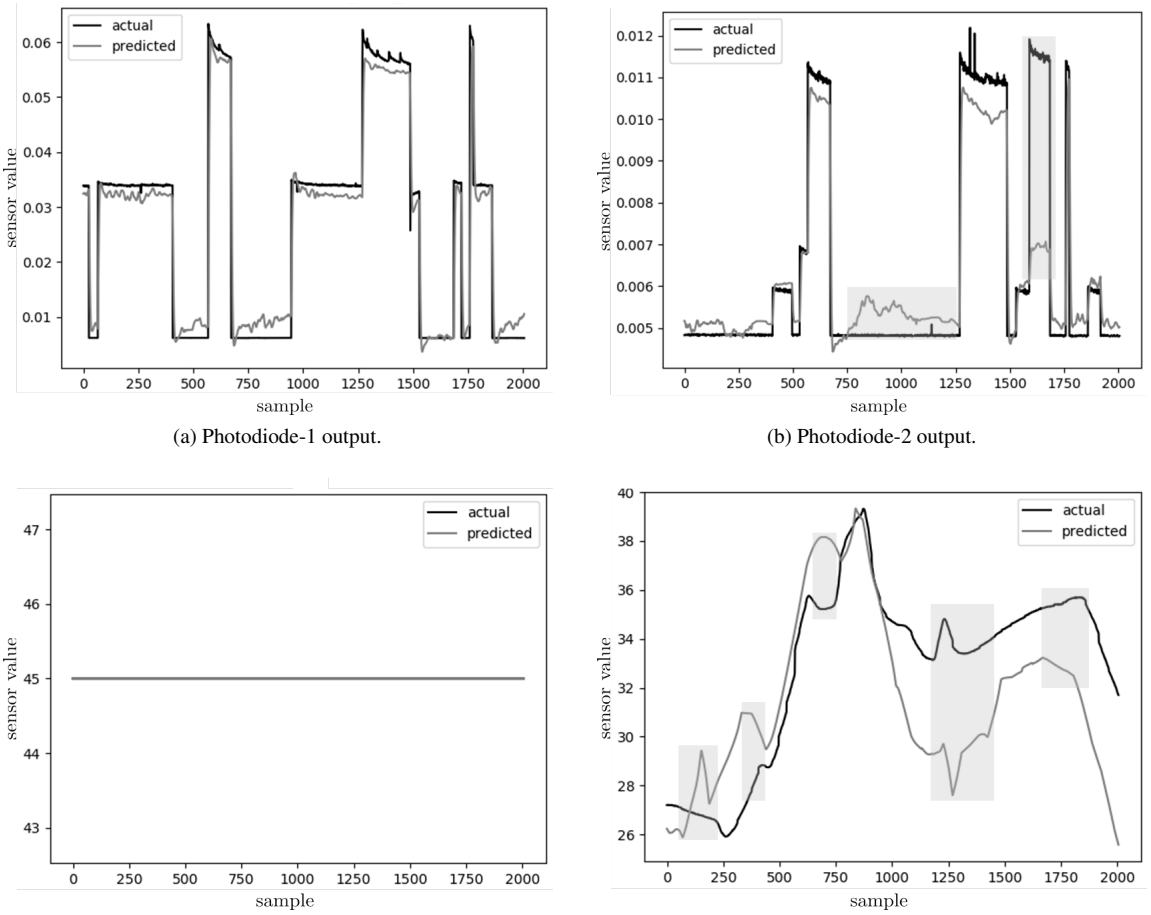


Fig. 5 Screenshot of the simulator driven by the LSTM model

VI. Empirical Results and Discussion

Predictions made by the LSTM model on the test set are shown in Fig. 6. Signals plotted in black are ground truth signals collected from the MAIUS hardware and the ones drawn in gray are predicted by the LSTM network. Regions in the plot, where the predicted signal deviates heavily from the true trajectory, are marked with a shaded background.



(a) Photodiode-1 output.

(b) Photodiode-2 output.

(c) Rotatory encoder output.

(d) Temperature sensor output.

Fig. 6 Predictions made by the LSTM model are shown in gray. Ground truth signals are drawn in black. Regions where the predicted signal deviates heavily from the true trajectory are marked with a shaded background.

Average values of the error metrics for all four signals are listed in Table 2. From the error values, one can see that predicting a constant signal regardless of the input is very easy for the model. The MAPE and sMAPE both are zero in case of the rotatory encoder. The average error is highest for the photodiode signals (20.6% MAPE for photodiode-2). This is mainly due to the nature of the signal. Since the photodiode outputs are step-like, in the region of rising and falling edges, the time lag in prediction by only one sample causes large errors in the absolute distance measure, hence increasing the overall error percentage.

Table 2 Average values of the error metrics

Metric	Photodiode-1	Photodiode-2	Rotatory Encoder	Temperature sensor
MAPE (%)	8.8	20.6	0.0	7.8
sMAPE (%)	9.0	16.0	0.0	8.1

Furthermore, we used the developed simulator application to manually test the behavior of the LSTM model. We could see that predictions for the temperature sensor, the rotatory encoder, and the photodiode-1 were as expected. However, in the case of photodiode-2, we discovered that when the voltage level of led-2 is changed, the photodiode-2 readings are also affected. Since, we used a divider to block the LED light (see Fig. 3), it is possible that the blocking was unable to block all light from led-2, so it also affected the photodiode-2 readings. In order to find the cause of this behavior, we manually inspected the training data and discovered that in some part of the experiment the photodiode-2 readings are indeed affected by led-2. This is caused by the external interference during the experiment. The LSTM is able to detect this interference and learned it as a normal behavior. Nevertheless, such a manual inspection of a system is only possible when the system is well known.

Moreover, when the input is not changed for a long period of time (more than 400 prediction cycles), all predictions saturate at a value. This is due to the shortcoming of RNNs in general that they cannot hold past information for an arbitrarily long period of time. Even LSTMs have difficulties in remembering information when the event has occurred in the distant past. We observed that this duration is related to the sequence length of the LSTM (see Sec. V.B). If this kind of behavior is not desired in the simulation, additional measures can be taken, for example, one can ensure that all actuator states are changed at least every 400 prediction cycles. With a prediction frequency of 0.5 hertz, 400 prediction cycles require 200 seconds. Another option could be to retrain the model with a longer sequence length. However, training an LSTM with a longer sequence length increases the depth of the network, which in return increases the risk of numerical instabilities in the network.

Nonetheless, for the use case of applying the model for testing the communication channels of an OBSW instead of mock-up methods, obtained results are promising. The proposed black-box modeling technique is able to model the system's dynamic behavior with an accuracy of about 80% on average. In case of using mock-up methods, the channel values are either random or constant. With the proposed method, the predicted channel values are likely to be within a tolerance of 20% from the ground truth.

VII. Conclusion and Outlook

The proposed ML technique allows fast creation and deployment of system models from the input/output data space of the system without having to acquire any domain-specific knowledge. From the empirical results, it is seen that RNN-based models are capable of generalizing different physical processes. Without any feature engineering, the proposed LSTM model is able to map feature deficient binary signals with a more complex continuous signal with an average error rate of about 8%. The model also shows good filtering capability by producing a constant value continuously regardless of the provided input. The only time-consuming task during development is the determination of the optimum hyperparameter set for the model which properly fits the given dataset and simulates the underlying physics. Nevertheless, once this is achieved, the LSTM model is able to model the hardware behavior and produce realistic outputs.

These kinds of data-driven models are most useful in case of highly complex, nonlinear, dynamic systems, where simplistic knowledge-based models are going to be either very expensive or technically unfeasible. A candidate system for such a case could be the payload hardware of complex space missions like MAIUS and BECCAL. Nevertheless, the next step of this research is to extend the proposed modeling technique to include a larger system incorporating several physical processes. Thus, future work could include exploring the scalability of the proposed modeling technique by applying it to a larger scientific dataset.

References

[1] Tamaskar, S., Neema, K., and DeLaurentis, D., "Framework for measuring complexity of aerospace systems," *Research in Engineering Design*, Vol. 25, 2014, pp. 125–137. https://doi.org/10.1007/s00163-014-0169-5.

[2] Kim, D., Park, S.-Y., Kim, J.-W., and Choi, K.-H., "Development of a Hardware-In-Loop (HIL) Simulator for Spacecraft Attitude Control Using Momentum Wheels," *Journal of Astronomy and Space Sciences*, Vol. 25, 2008, pp. 347–360. https://doi.org/10.5140/JASS.2008.25.4.347.

[3] Jung, D., and Tsiotras, P., "Modeling and Hardware-in-the-Loop Simulation for a Small Unmanned Aerial Vehicle," *American Institute of Aeronautics and Astronautics (AIAA)*, California, USA, 2007, pp. 1–3.

[4] Yu, M., Tang, X., Lin, Y., and Wang, X., "Diesel Engine Modeling based on Recurrent Neural Networks for a Hardware-in-the-Loop Simulation System of Diesel Generator Sets," *Neurocomputing*, Vol. 283, 2018, pp. 9–19. https://doi.org/https://doi.org/10.1016/j.neucom.2017.12.054, URL http://www.sciencedirect.com/science/article/pii/S0925231217319161.

[5] Kumarin, A., Kuznetsov, A., and Makaryants, G., "Hardware-in-the-loop Neuro-based Simulation for Testing Gas Turbine Engine Control System," *Global Fluid Power Society PhD Symposium (GFPS)*, Samara, Russia, 2018, pp. 1–5.

[6] Khasnabish, N., Suggu, D., and Koppad, A., "Neural Network Based Throttle Actuator Model for Controller," *Symposium on International Automotive Technology*, Pune, India, 2019, pp. 1–6.

[7] Lipton, Z. C., Kale, D. C., Elkan, C., and Wetzel, R. C., "Learning to Diagnose with LSTM Recurrent Neural Networks," *International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, 2016, pp. 1–18.

[8] White, G., Palade, A., and Clarke, S., "Forecasting QoS Attributes Using LSTM Networks," *International Joint Conference on Neural Networks (IJCNN)*, Rio, Brazil, 2018, pp. 1–9.

[9] Hochreiter, S., and Schmidhuber, J., "Long Short-Term Memory," *Journal of Neural Computation*, Vol. 9, No. 8, 1997, pp. 1735–1780.

[10] Cho, K., v. Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y., "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar, 2014, pp. 1724–1734.

[11] Luo, C., Yang, D., Huang, J., and Deng, Y.-D., "LSTM-Based Temperature Prediction for Hot-Axles of Locomotives," *International Conference on Information Technology and Applications (ITA)*, Beijing, China, 2017, pp. 1–6.

[12] Zhang, Q., Wang, H., Dong, J., Zhong, G., and Sun, X., "Prediction of Sea Surface Temperature using Long Short-Term Memory," *Journal of IEEE Geoscience and Remote Sensing Letters*, Vol. 14, No. 10, 2017, pp. 1745–1749.

[13] Hundman, K., Constantinou, V., Laporte, C., Colwell, I., and Soderstrom, T., "Detecting Spacecraft Anomalies Using LSTMs and Nonparametric Dynamic Thresholding," *CoRR*, Vol. abs/1802.04431, 2018. URL http://arxiv.org/abs/1802.04431.

[14] Liu, H., Liu, C., Wang, J., and Wang, H., "Predicting Solar Flares Using a Long Short-term Memory Network," *The Astrophysical Journal*, Vol. 877, No. 2, 2019, pp. 1–19. https://doi.org/10.3847/1538-4357/ab1b3c.

[15] Fuertes, S., Picart, G., Tourneret, J.-Y., Chaari, L., Ferrari, A., and Richard, C., "Improving Spacecraft Health Monitoring with Automatic Anomaly Detection Techniques," *SpaceOps Conference*, Daejeon, Korea, 2016, pp. 1–16. https://doi.org/10.2514/6.2016-2430.

[16] Bobra, M. G., and Ilonidis, S., "Predicting Coronal Mass Ejections Using Machine Learning Methods," *The Astrophysical Journal*, Vol. 821, No. 2, 2016, pp. 1–16. https://doi.org/10.3847/0004-637x/821/2/127.

[17] Liu, Q., Basu, S., Ganguly, S., Mukhopadhyay, S., DiBiano, R., Karki, M., and Nemani, R., "DeepSat V2: Feature Augmented Convolutional Neural Nets for Satellite Image Classification," *Remote Sensing Letters*, Vol. 11, No. 2, 2019, pp. 156–165. https://doi.org/10.1080/2150704x.2019.1693071.

[18] Dey, T. K., Wang, J., and Wang, Y., "Road Network Reconstruction from satellite images with Machine Learning Supported by Topological Methods," *International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, 2019. https://doi.org/10.1145/3347146.3359348.

[19] Ullo, S., Langenkamp, M., Oikarinen, T., DelRosso, M., Sebastianelli, A., Piccirillo, F., and Sica, S., "Landslide Geohazard Assessment with Convolutional Neural Networks Using Sentinel-2 Imagery Data," *IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, Yokohama, Japan, 2019, pp. 1–4. https://doi.org/10.1109/IGARSS.2019.8898632.

[20] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning representations by back-propagating errors," *Neurocomputing: Foundations of Research*, MIT Press, Cambridge, Massachusetts, USA, 1988, pp. 696–699.

[21] Lipton, Z. C., Berkowitz, J., and Elkan, C., "A Critical Review of Recurrent Neural Networks for Sequence Learning," *CoRR*, Vol. abs/1506.00019, 2015, pp. 1–38.

[22] Hochreiter, S., Bengio, Y., and Frasconi, P., "Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies," *Field Guide to Dynamical Recurrent Networks*, IEEE Press, New Jersey, USA, 2001, pp. 1–15.

[23] Bengio, Y., Simard, P., and Frasconi, P., "Learning Long-Term Dependencies with Gradient Descent is Difficult," *Journal of IEEE Transactions on Neural Networks*, Vol. 5, No. 2, 1994, pp. 157–166.

[24] Pascanu, R., Mikolov, T., and Bengio, Y., "On the difficulty of training recurrent neural networks," *International Conference on Machine Learning (ICML)*, Atlanta, USA, 2013, pp. 1–7.

[25] Gers, F. A., Schmidhuber, J., and Cummins, F. A., "Learning to Forget: Continual Prediction with LSTM," *Journal of Neural Computation*, Vol. 12, No. 10, 2000, pp. 2451–2471.

[26] Graves, A., "Generating Sequences With Recurrent Neural Networks," *Journal of CoRR*, Vol. 1308, No. 850, 2013, pp. 157–166.

[27] Becker, D., Lachmann, M. D., Seidel, S. T., Ahlers, H., Dinkelaker, A. N., Grosse, J., Hellmig, O., Müntinga, H., Schkolnik, V., Wendrich, T., Wenzlawski, A., Weps, B., Corgier, R., Franz, T., Gaaloul, N., Herr, W., Lüdtke, D., Popp, M., Amri, S., Duncker, H., Erbe, M., Kohfeldt, A., Kubelka-Lange, A., Braxmaier, C., Charron, E., Ertmer, W., Krutzik, M., Lämmerzahl, C., Peters, A., Schleich, W. P., Sengstock, K., Walser, R., Wicht, A., Windpassinger, P., and Rasel, E. M., "Space-borne Bose–Einstein condensation for precision interferometry," *Nature*, Vol. 562, No. 7727, 2018, p. 391–395. https://doi.org/10.1038/s41586-018-0605-1.

[28] Weps, B., Lüdtke, D., Franz, T., Maibaum, O., Wendrich, T., Müntinga, H., and Gerndt, A., "A Model-driven Software Architecture for Ultra-cold Gas Experiments in Space," *International Astronautical Congress (IAC)*, Bremen, Germany, 2018, pp. 1–10.

[29] Frye, K., Abend, S., Bartosch, W., Bawamia, A., Becker, D., Blume, H., Braxmaier, C., Chiow, S.-W., Efremov, M. A., Ertmer, W., Fierlinger, P., Gaaloul, N., Grosse, J., Grzeschik, C., Hellmig, O., Henderson, V. A., Herr, W., Israelsson, U., Kohel, J., Krutzik, M., Kürbis, C., Lämmerzahl, C., List, M., Lüdtke, D., Lundblad, N., Marburger, J. P., Meister, M., Mihm, M., Müller, H., Müntinga, H., Oberschulte, T., Papakonstantinou, A., Perovšek, J., Peters, A., Prat, A., Rasel, E. M., Roura, A., Schleich, W. P., Schubert, C., Seidel, S. T., Sommer, J., Spindeldreier, C., S.-K., D., Stuhl, B. K., Warner, M., Wendrich, T., Wenzlawski, A., Wicht, A., Windpassinger, P., Yu, N., and Wörner, L., "The Bose-Einstein Condensate and Cold Atom Laboratory," *arXiv*, 2019, pp. 1–37. URL https://arxiv.org/abs/1912.04849v1.

[30] Shanker, M., Hu, M. Y., and Hung, M. S., "Effect of Data Standardization on Neural Network Training," *Omega*, Vol. 24, No. 4, 1996, pp. 385–397.

[31] Bengio, Y., "Practical Recommendations for Gradient-based Training of Deep Architectures," *Neural Networks: Tricks of the Trade: Second Edition*, Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 437–478. https://doi.org/10.1007/978-3-642-35289-8.

[32] Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B., "Algorithms for Hyper-parameter Optimization," *24th International Conference on Neural Information Processing Systems (NIPS)*, Granada, Spain, 2011, pp. 2546–2554.

[33] Ioffe, S., and Szegedy, C., "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *International Conference on Machine Learning (ICML)*, Lille, France, 2015, pp. 448–456.

[34] Xu, B., Wang, N., Chen, T., and Li, M., "Empirical Evaluation of Rectified Activations in Convolutional Network," *CoRR*, Vol. abs/1505.00853, 2015, pp. 1–7. URL http://arxiv.org/abs/1505.00853.

[35] Nair, V., and Hinton, G., "Rectified Linear Units Improve Restricted Boltzmann Machines," *International Conference on Machine Learning (ICML)*, Vol. 27, 2010, pp. 807–814.

[36] LeCun, Y., Bengio, Y., and Hinton, G., "Deep learning," *Nature*, Vol. 521, 2015, pp. 436–444.

[37] Kingma, D. P., and Ba, J. L., "Adam: A Method for Stochastic Optimization," *IEEE International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, 2015, pp. 1–15.

[38] Glorot, X., and Bengio, Y., "Understanding the difficulty of training deep feedforward neural networks," *International Conference on Artificial Intelligence and Statistics (AISTATS)*, Sardinia, Italy, 2010, pp. 249–256.