36th European Modeling & Simulation Symposium, 035
21st International Multidisciplinary Modeling & Simulation Multiconference

2724-0029 © 2024 The Authors.
doi: 10.46354/i3m.2024.emss.035

Explaining LSTM neural networks: A DEVS-based approach

Ali Ayadi[1,*], Thi-My-Duyen Truong[2], Claudia Frydman[2] and Cédric Wemmert[1]

[1]SDC team, ICube UMR 7357 CNRS, Université de Strasbourg, Illkirch, France
[2]MOFED team, LIS UMR 7020 CNRS, Aix Marseille Université, Marseille, France

[*]Corresponding author. Email address: ali.ayadi@unistra.fr

Abstract

Long Short-Term Memory (LSTM) neural networks are important for processing sequential data in various applications, from natural language processing to time series analysis. Their interpretability is limited due to intricate internal mechanisms. This paper addresses these limiblack boxesenhancing LSTMs interpretability through the Discrete Event System Specification (DEVS) formalism. We model each LSTM component as distinct DEVS models, creating a modular and hierarchical framework that allows detailed visualizations of data flow and memory updates to support step-by-step simulation and analysis of LSTM operations over time. We developed a DEVS-based LSTM library in the DEVSimPy environment for modular testing and simulation. We demonstrate the effectiveness of our DEVS-based framework through a proof of concept, where the library's capability to replicate known LSTM behaviors is validated against manually computed outcomes. This case study elucidates the operations within LSTM and enhances their interpretability, bridging the gap between complex LSTM operations and user comprehensibility.

Keywords: Long Short-Term Memory; Discrete Event System Specification; Explainable AI; Interpretability.

1. Introduction

LSTM networks are sophisticated recurrent neural networks (RNNs) designed to process and model sequential data (Sherstinsky, 2020). Developed in the 1990s, LSTMs address the critical challenges faced by traditional RNNs, particularly their inability to capture long-range dependencies due to the vanishing gradient problem (Louis, 2024). This enhancement allows LSTMs to perform exceptionally well in applications such as natural language processing, speech recognition, and time series forecasting, where understanding extended sequences is crucial (Torres et al., 2021).

However, despite their performance, LSTMs often operate as black boxes, with limited transparency regarding how they process inputs to produce outputs (Buhrmester et al., 2021). This lack of interpretability poses significant challenges, particularly in domains where understanding the decision-making process is essential, such as in regulatory and safety-critical environments (Saraswat et al., 2022).

In the context of the growing demand for explainable artificial intelligence (XAI), enhancing the transparency of LSTM models is more than just a technical improvement - it's a necessary improvement to meet regulatory and ethical standards. XAI seeks to make machine learning models accessible and understandable to human stakeholders, thus promoting trust and facilitating more informed decision-making (Islam et al., 2022). As AI systems become more integrated into critical decision processes in industries, such as healthcare, finance, and legal systems, the ability to interpret and verify the reasoning behind AI decisions becomes imperative. By integrating Discrete Event System Specification (DEVS) formalism into LSTM

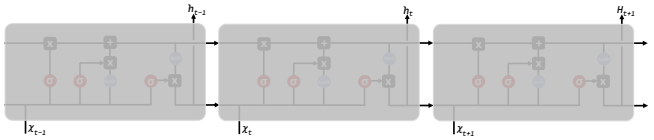

Figure 1. Three-module chain of LSTM networks.

networks, this work contributes to the field of XAI by providing a methodological framework that explains how these models process data and simplifies their complexity into interpretable components. This advancement aids stakeholders in assessing the reliability and fairness of AI-driven decisions, aligning LSTM applications with the principles of responsible AI usage.

To address these challenges, this paper introduces the application of the DEVS formalism to LSTM networks. DEVS offers a structured, modular approach that enhances the transparency and comprehensibility of complex systems (Alshareef et al., 2022). By modeling each LSTM component through the DEVS framework, we aim to provide a clear, step-by-step visualization of the internal dynamics of LSTM operations, thereby demystifying their functionality and making these powerful models more accessible and understandable.

2. Preliminaries

2.1. Long Short-Term Memory

LSTM networks are designed to address the vanishing gradient problem commonly encountered in training neural networks to learn long-term dependencies (Anbil Parthipan, 2020). The architecture features a constant error carousel (CEC) that stabilizes the error signal within each cell, effectively preventing gradient issues (Kisvari et al., 2021). Each LSTM cell incorporates components such as input, output, and forget gates that manage the flow of information (Kisvari et al., 2021).

Architecture. As shown in Figure 1, the LSTM architecture comprises multiple recurrently linked memory blocks, each designed to maintain its state over time and control information flow through nonlinear gating mechanisms (Van Houdt et al., 2020). Figure 2 displays the structure of a typical LSTM memory cell, which includes the gates and activation functions that facilitate the processing of inputs and generation of outputs.

Operational Dynamics. The operational dynamics of a LSTM network involve several key stages, each defined by specific equations, to update the state of the network.

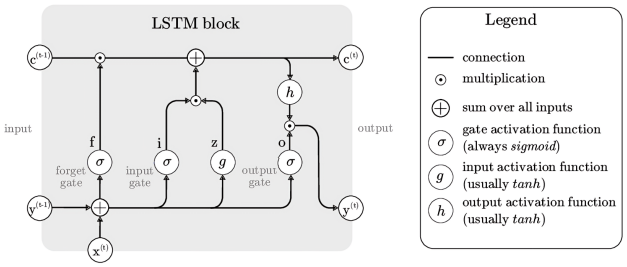

Figure 2. Architecture of a typical LSTM block (Van Houdt et al., 2020).

Below are the main operations performed in an LSTM unit:

$$z(t) = g(W_z x(t) + R_z y(t-1) + b_z) \quad (1)$$

$$i(t) = \sigma(W_i x(t) + R_i y(t-1) + p_i \cdot c(t-1) + b_i) \quad (2)$$

$$f(t) = \sigma(W_f x(t) + R_f y(t-1) + p_f \cdot c(t-1) + b_f) \quad (3)$$

$$c(t) = z(t) \cdot i(t) + c(t-1) \cdot f(t) \quad (4)$$

$$o(t) = \sigma(W_o x(t) + R_o y(t-1) + p_o \cdot c(t) + b_o) \quad (5)$$

$$y(t) = g(c(t)) \cdot o(t) \quad (6)$$

Equations 1 to 6 depict how each component of the LSTM cell interacts with others to process data, computing the block input, input gate, forget gate, candidate cell memory, output gate, and the output block. This setup facilitates understanding LSTM mechanisms and underscores the model's capability in handling sequential data effectively.

2.2. DEVS and DEVSimPy Environment

The DEVS (Discrete Event System Specification) formalism offers a modular framework for discrete event systems, enhancing complex system analysis and design with features like completeness and extensibility. It distinguishes between *atomic models*, which detail operational dynamics, and *coupled models*, which integrate multiple models for complex systems (Kim and Zeigler, 1987).

Atomic models are defined by a 7-tuple indicating inputs, outputs, states, and functions, while coupled models manage interactions and choose between conflicting events with a SELECT function. Recent tools have built on DEVS to provide advanced modeling capabilities (The-RED-Network, 2024).

For our applications, we utilize DEVSimPy, an open-source platform that extends DEVS into Python, facilitating complex model integration within a user-friendly environment (Capocchi et al., 2011). Developed by the *Science pour l'Environnement* team at the University of Corsica, DEVSimPy supports extensive libraries and provides a unified platform for discrete event and ANN modeling, ensuring streamlined model development and simulation.

3. State of the art

Explainability in ANN is a crucial area in AI that aims to make machine and deep learning systems more understandable (Samek et al., 2017). While ANN excel at tasks such as image recognition or language understanding, their decision-making processes are often opaque, leading to their characterization as black boxes. To address this, researchers have been integrating machine learning with domain-specific knowledge to create more transparent models (Roscher et al., 2020; Tiddi and Schlobach, 2022; Saeed and Omlin, 2023). Despite these efforts, no approach has fully overcome the challenges posed by the complex architectures of ANN, which often hinder attempts to enhance transparency without degrading performance.

In response, some developers have turned to the DEVS formalism as a promising solution due to its inherent transparency, modularity, and capability to decompose complex systems. This approach could significantly improve understanding and trust in AI systems. For instance, (Toma et al., 2011) were the first to propose a new modeling approach for ANNs using DEVS. This method aims to simplify the configuration and training phases of ANNs by making them more modular and comprehensible. By decomposing the ANN into atomic and coupled models, the authors effectively manage the inherent complexity associated with ANN design. Their validation with a non-linearly separable two-dimensional XOR problem demonstrates the feasibility and effectiveness of DEVS for ANN modeling, providing a clear insight into the network's architecture and behavior beyond the typical black-box perception. (Sehili et al., 2015) explore the integration of DEVS formalism with ANN models to enhance transparency and modularity in simulations. Their study applies these techniques across several case studies, showcasing how DEVS can offer a structured, systematic approach to understanding and modifying ANN models. The methodology they develop for decomposing complex ANN architectures into simpler, discrete event models makes these systems easier to analyze and adapt. (Adelani and Traoré, 2016) investigate using DEVS to optimize ANN training processes. They propose a hybrid model that combines traditional ANN training algorithms with discrete event simulation, enhancing training efficiency and scalability. Their comparative analysis illustrates that this approach can significantly reduce computational demands and improve scalability, especially in scenarios involving large datasets and complex network architectures.

However, to the best of our knowledge, no research has yet explored applying DEVS to understand LSTM networks. This gap presents a unique opportunity for our study.

4. LSTM DEVS-based modeling approach

4.1. Overall model structure

As a top-down approach for effective description, we begin by outlining the structure of the proposed DEVS-based LSTM model, as depicted in Figure 3a, featuring three key layers. The `Input_layer_AM`, an atomic model, processes raw inputs are processed into a format suitable for further analysis. The `Hidden_layer_CM`, a coupled model, contains nine interconnected LSTM cells (`LSTM_cell_CM_1` to `LSTM_cell_CM_9`) that manage temporal dependencies, updating and maintaining information to capture dynamic input attributes. Then, the `Output_layer_AM`, another atomic model, synthesizing the processed data into final outputs for predictions or classifications. This architecture highlights the sequential data management capabilities of LSTM networks through detailed DEVS modeling.

This architecture presented in Figure 3a is structured by the `LSTM_Network_CM` coupled model, covering the entire LSTM network from data input to output. This model includes three main components: the `Input_layer_AM`, `Hidden_layer_CM`, and `Output_layer_AM`, which ensure a smooth flow of data processing. Data is initially processed by the `Input_layer_AM`, managed for temporal dependencies by the `Hidden_layer_CM`, and finally synthesized into actionable outputs by the `Output_layer_AM`. The model's internal and external couplings facilitate data transfer and interaction with the environment. A selection function within the model prioritizes events to manage responses from multiple sub-models. This `LSTM_Network_CM` is defined by the following formal specification:

Formal specification of the `LSTM_Network_CM`:

X = {input}, external sequences for processing
Y = {output}, final decision or prediction
$\{M_i\}$ = {Input_layer_AM, Hidden_layer_CM, Output_layer_AM}
EIC = {(CM.input, Input_layer_AM.input)}
EOC = {(Output_layer_AM.output, CM.output)}
IC = {(Input_layer_AM.output, Hidden_layer_CM.input), (Hidden_layer_CM.output, Output_layer_AM.input)}
Select = function prioritizing internal model events

4.2. Input layer atomic model

The `Input_layer_AM` of an LSTM neural network is crucial for the initial processing of input data sequences. As an atomic model, it receives raw data, transforming it into a format suitable for subsequent network layers. The model's formal specifications detail its functionality: inputs *X* include any `data` (sequences or vectors), while outputs *Y* produce `processed_data`, tailored for the hidden layers. Operational states of the `Input_layer_AM` are *idle* and *processing*. Starting in *idle*, it transitions to *processing* upon data reception, as governed by the external transition function δ_{ext}. Data transformation occurs during this phase, with the duration set by `processing_time` in the time advance function *ta*. After processing, the model outputs formatted data and reverts to *idle* via the internal transition function δ_{int}. Output generation, controlled by

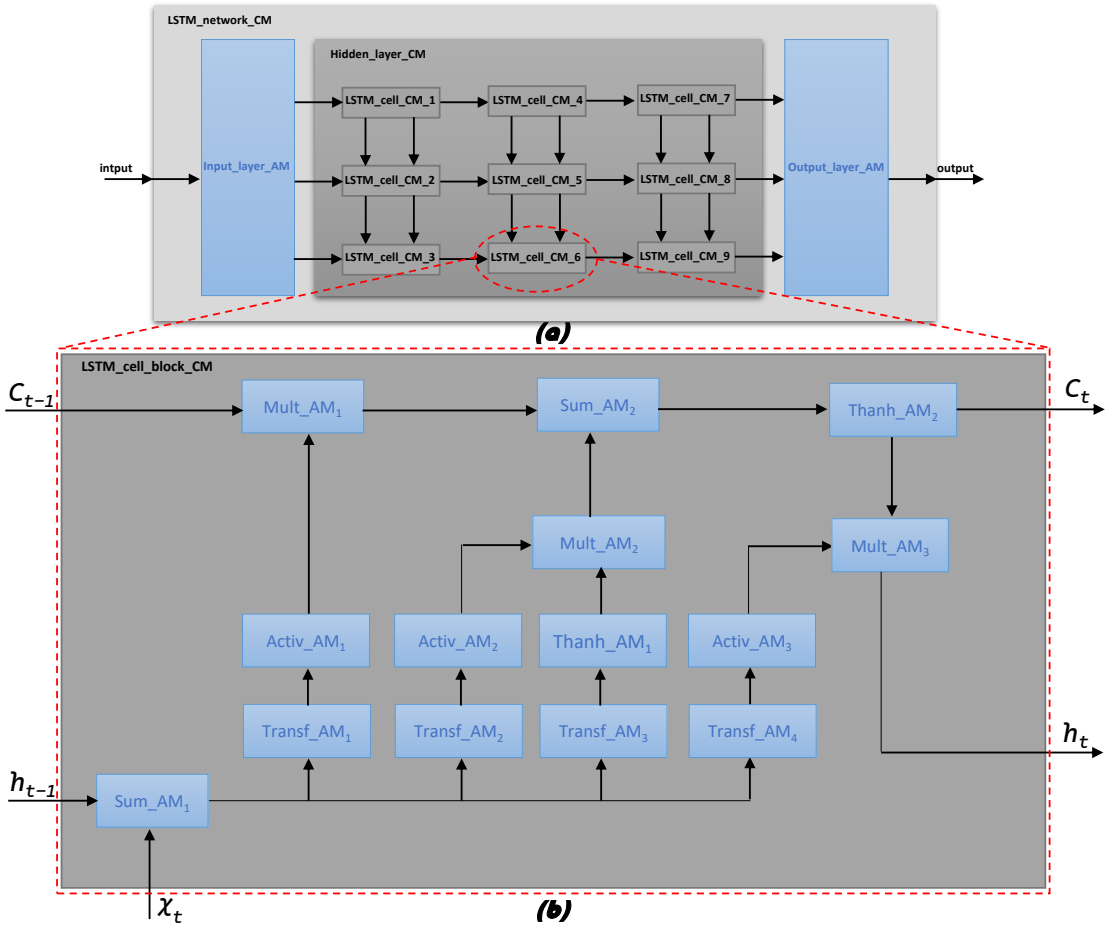

Figure 3. Overall DEVS model structure of: (a) an LSTM neural network, featuring an input layer atomic model, a hidden layer coupled model with nine single-cell blocks, and an output layer atomic model; and a (b) detailed view of an LSTM cell block.

the output function λ, occurs only in *processing* state, ensuring efficient data preparation for deeper LSTM network processing.

Formal specification of the `Input_layer_AM`:

X = {data}
Y = {processed_data}
S = {"idle", "processing"}
s_0 = "idle"
ta("idle") = ∞
ta("processing") = processing_time
δ_{ext}("idle", e, {data}) = "processing"
δ_{int}("processing") = "idle"
λ("processing") = {processed_data}
where processed_data is the transformed input data.

4.3. Hidden layer coupled model

The `Hidden_layer_CM` coupled model functions as the central processing unit of the LSTM network, containing a series of LSTM memory cells (from `LSTM_cell_CM_1` to `LSTM_cell_CM_9`) that handle detailed computations of temporal data. It receives processed inputs from the `Input_layer_AM`, which are then sequentially processed by interconnected LSTM memory cells, each contributing uniquely to the learning process. Outputs from certain cells, such as `LSTM_cell_CM_7` to `LSTM_cell_CM_9`, are designated as outputs for the entire hidden layer, which are then forwarded to the `Output_layer_AM` for final processing. The selection function here manages internal conflicts and scheduling, ensuring that the most critical processing tasks are prioritized.

Formal specification of the `Hidden_layer_CM`:

X = {processed_input}, data from Input_layer_AM
Y = {hidden_output}, data for Output_layer_AM
{M_i} = {LSTM_cell_CM_1, LSTM_cell_CM_2, ...,
LSTM_cell_CM_9}
EIC = {(CM.processed_input, LSTM_cell_CM_1.input)}
EOC = {(LSTM_cell_CM_7.output, CM.hidden_output)}
IC = {(LSTM_cell_CM_1.output, LSTM_cell_CM_4.input),
(LSTM_cell_CM_4.output, LSTM_cell_CM_7.input)}
Select = function managing outputs from LSTM cells

4.4. LSTM memory cell coupled model

As illustrated in Figure 3b, the heart of the LSTM network is the `LSTM_cell_block_CM` coupled model, which is composed of multiple atomic models including *Multiplication Units* (`Mult_AM`) for critical element-wise multiplications essential for gate modulation, *Summation Units* (`Sum_AM`) which aggregate various inputs to update the cell's state, and *Transformation Modules* (`Transf_AM`) that apply nonlinear activations to enable the learning of complex patterns. Additional *Functional Modules* such as `Than_AM` employ hyperbolic tangent function for regulating the flow of information within the cell.

Formal specification of the `LSTM_cell_block_CM`:

X = {input_signals}, inputs to the LSTM cell
Y = {output_signals}, outputs from the LSTM cell
{M_i} = {Sum_AM1, Transf_AM1, activ_AM1, Mult_AM1, Transf_AM2, activ_AM2, Mult_AM3, Sum_AM2, FM_Thanh_AM1, Transf_AM3, Mult_AM4, activ_AM3, Transf_AM4, FM_Thanh_AM2}
EIC = {(CM.input_signals, Sum_AM1.input), (CM.input_signals, Transf_AM1.input)}
EOC = {(Mult_AM4.output, CM.output_signals)}
IC = {(Sum_AM1.output, Mult_AM1.input), (Transf_AM1.output, activ_AM1.input), (activ_AM1.output, Mult_AM1.input), (Mult_AM1.output, Sum_AM2.input), (Transf_AM2.output, activ_AM2.input), (activ_AM2.output, Mult_AM3.input), (Mult_AM3.output, Sum_AM2.input), (Sum_AM2.output, FM_Thanh_AM1.input), (FM_Thanh_AM1.output, Transf_AM3.input), (Transf_AM3.output, Mult_AM4.input), (Transf_AM4.output, activ_AM3.input), (activ_AM3.output, FM_Thanh_AM2.input), (FM_Thanh_AM2.output, Mult_AM4.input)}
Select = function managing the prioritization of simultaneous events

4.4.1. Multiplication atomic model

The atomic model of multiplication, designated as `Mult_AM1`, `Mult_AM2`, and `Mult_AM3`. Each one of these three atomic models processes a distinct pair of real numbers crucial for different phases of the LSTM's computational flow:

- `Mult_AM1` multiplies the current state cell $C(t)$ with the output from the forget gate. This updates the state cell based on past information, while ensuring that irrelevant data is forgotten.
- `Mult_AM2` multiplies the candidate state $z(t)$ with the output of the input gate. This integrates new input information into the state cell, allowing the LSTM to learn from new data.
- `Mult_AM3` multiplies the updated state cell $C(t)$ by the output of the output gate to calculate the final hidden state output $h(t)$. This last multiplication defines what the next or output layer of the neural network receives from the LSTM memory cell.

Inputs X for the multiplication model, `Mult_AM1`, `Mult_AM2`, and `Mult_AM3`, consist of real number pairs $(v1, v2)$ to be multiplied. These pairs may originate from various LSTM operational parameters like activation function outputs or state cell values. The output Y is a real number w, the product of the inputs, useful for further computations or feedback mechanisms in larger systems. All three models share identical operational dynamics within the DEVS framework. The model alternates between two primary states: *idle* and *calculate*. It remains in *idle* indefinitely until inputs are received, then instantly transitions to *calculate* to perform the multiplication. The time advance function $ta()$, external transition function δ_{ext}, and internal transition function δ_{int} govern these state changes and processing times, ensuring immediate responsiveness and rapid return to *idle* after calculations. The output function λ emits the result instantaneously once the computation is complete.

Formal specification of the `Mult_AM`:

X = {(v1, v2) | $v1 \in \mathbb{R}, v2 \in \mathbb{R}$}, where for:
  Mult_AM1: X1 = {(C(t), fg) | $C(t) \in \mathbb{R}, fg \in \mathbb{R}$}
  where: C(t) the current state cell value,
    and fg the output from the forget gate
  Mult_AM2: X2 = {(z(t), ig) | $z(t) \in \mathbb{R}, ig \in \mathbb{R}$}
  where: z(t) the candidate state,
    ig the output from the input gate
  Mult_AM3: X3 = {(C(t), og) | $C(t) \in \mathbb{R}, og \in \mathbb{R}$}
  where: C(t) the updated state cell value,
    og the output from the output gate
Y = {w | $w \in \mathbb{R}$}, where for:
  Mult_AM1: Y1 = {w1 | $w1 \in \mathbb{R}$}
  Output w1 is the product of C(t) and fg.
  Mult_AM2: Y2 = {w2 | $w2 \in \mathbb{R}$}
  Output w2 is the product of z(t) and ig.
  Mult_AM3: Y3 = {w3 | $w3 \in \mathbb{R}$}
  Output w3 is the product of C(t) and og.
S = {"calculate", "idle"},
  with s0 = "idle"
ta("calculate") = 0, ta("idle") = ∞
δ_{ext}("idle", e, ($v1$, $v2$)) = "calculate",
δ_{ext}("calculate", e, ($v1$, $v2$)) = "calculate"
δ_{int}("calculate") = "idle"
λ("calculate") = $v1 \times v2$

4.4.2. Summation atomic model

The summation atomic model (`Sum_AM`) is designed to perform essential summation operations necessary for updating the LSTM's internal states. We distinguish two types:

- `Sum_AM1`: It sums the results obtained by the forget gate and the input gate with their respective parts of the state cell. This summation updates the state cell to its new state $C(t)$, combining both past information and

new incoming information.
- `Sum_AM2`: This model sums the product of the updated state cell and the output from the output gate to calculate the final output state $h(t)$.

The `Sum_AM` atomic models operate in two states: *calculate* for active processing and *idle* for waiting. The δ_{ext} function switches the model from *idle* to *calculate* upon receiving inputs, while the δ_{int} function resets the state to *idle* after processing. The λ function computes the sum of the inputs during the *calculate* state. Time advancement is immediate in the *calculate* state and infinite in the *idle* state, indicating that the model waits indefinitely until the next input arrives.

Formal specification of the `Sum_AM`:

X = {(v1, v2) | v1 $\in \mathbb{R}$, $v2 \in \mathbb{R}$}, where for:
  Sum_AM1: X1 = {(C(t), fg) | $C(t) \in \mathbb{R}, fg \in \mathbb{R}$}
  where: C(t) is the current state cell value,
  and fg is the output from the forget gate
  Sum_AM2: X2 = {(z(t), ig) | $z(t) \in \mathbb{R}, ig \in \mathbb{R}$}
  where: z(t) is the candidate state,
  and ig is the output from the input gate
Y = {s | $s \in \mathbb{R}$}, where for:
  Sum_AM1: Y1 = {s1 | $s1 \in \mathbb{R}$}
  Output s1 the sum of (C(t) $\times fg$) and ($z(t) \times ig$)
  Sum_AM2: Y2 = {s2 | $s2 \in \mathbb{R}$}
  Output s2 is the sum of (C(t) $\times og$)
S = {"calculate", "idle"}
  with s0 = "idle"
ta("calculate") = 0, ta("idle") = ∞
δ_{ext}("idle", e, (product)) = "calculate",
δ_{ext}("calculate", e, (product)) = "calculate"
δ_{int}("calculate") = "idle"
λ("calculate") = sum of inputs

4.4.3. *Transfer function atomic model*

The `Transf_AM` atomic model manages the linear transformation of inputs. It operates by applying a predetermined weight matrix and bias to the input data, which typically originates from previous layers or states within the LSTM architecture. The model starts in an *idle* state until it receives input data. Once input is available, the model switches to the *processing* state, where it performs the core operation of matrix multiplication followed by bias addition. In the *processing* state, the transformation is executed instantly. After the transformation operation, the model outputs the transformed data, which is usually passed on to an activation function to introduce non-linearity into the process. The temporal dynamics of the model are controlled by the time advance function, which is set to infinity in the *idle* state, indicating that the model will wait indefinitely for input. The transition to the *processing* state occurs instantaneously upon receiving input, and similarly, the model returns to the *idle* state immediately after processing the input. The δ_{ext} function dictates the model's response to incoming data, prompting a shift from waiting to active computation. Conversely, the δ_{int} function manages the model's return to readiness, resetting its state post-computation. The output function precisely defines the mathematical operation performed during the *processing* state, where the input vector is transformed by the weight matrix and adjusted by the bias, producing the output vector that is then ready for further processing stages in the network.

Formal specification of the `Transf_AM`:

X = {x | $x \in \mathbb{R}^n$}, x the input vector from previous layers
Y = {y | $y \in \mathbb{R}^m$}, y transformed input ready for activation
S = {"idle", "processing"} with $s0$ = "idle"
ta("idle") = ∞, **ta**("processing") = process time
δ_{ext}("idle", e, x) = "processing"
δ_{int}("processing") = "idle"
λ("processing") = $Wx + b$
  where W the weight matrix and b the bias vector

4.4.4. *Activation function atomic model*

The activation atomic model (`Activ_AM`) begins in the *idle* state, indicating it is prepared to receive inputs. The input set X consists of vectors that are the output of the `Transf_AM`, which has already applied linear transformations to the data. The activation function, represented in the model by $f(x)$, is applied when the model is in the *active* state, processing the data almost instantaneously to apply the non-linear transformation. After processing, the output function λ produces the activated output y, which is then either fed into further processing stages within the LSTM or used as part of the network's output. The δ_{ext} function handles the receipt of new input data by transitioning the model to the *active* state for processing. The δ_{int} function ensures the model returns to the *idle* state after activation, awaiting new inputs for subsequent activations. The time advance function confirms the instantaneous processing in the *active* state and indefinite waiting in the *idle* state until new data arrives.

Formal specification of the `Activ_AM`:

X = {x | $x \in \mathbb{R}^m$}
  the input vector x from the transformation model
Y = {y | $y \in \mathbb{R}^m$}
  the output vector y is the activated input
S = {"idle", "active"}
$s0$ = "idle"
ta("idle") = ∞, **ta**("active") = processing time
δ_{ext}("idle", e, x) = "active"
  the transition to active on receiving input
δ_{int}("active") = "idle"
  the transition back to ready after activation
λ("active") = $f(x)$
  the result of applying the activation function

4.5. Output layer atomic model

The `Output_layer_AM` in the LSTM network is the terminal stage, synthesizing computations from previous layers into actionable outcomes. As an atomic model, it receives arrays of hidden and cell states from the hidden layer, which contain the network's learned features. These are transformed into final outputs used for predictions—either as a probability distribution for classification via a softmax function or as continuous values for regression through linear transformations and activation functions.

Initially in an *idle* state, the `Output_layer_AM` transitions to a *generating_output* state upon receiving data. It processes the data into final outputs during a fixed `generation_time`. Once the output is issued, the model reverts to the *idle* state, maintaining an efficient data flow through the network.

Formal specification of the `Output_layer_AM`:

X = {processed_data_from_hidden_layer}
Y = {final_output}
S = {"idle", "generating_output"}
s_0 = "idle"
ta("idle") = ∞
ta("generating_output") = generation_time
δ_{ext}("idle", e, {processed_data_from_hidden_layer}) = "generating_output"
δ_{int}("generating_output") = "idle"
λ("generating_output") = {final_output}
where *final_output* is the prediction/classification result

5. Validation of the DEVS-based LSTM model

5.1. Development of the DEVS-based LSTM library

We developed a comprehensive library within the DEVSimPy environment to support the modeling and simulation of LSTM networks using the DEVS formalism. This library, meticulously programmed in Python, features a variety of atomic models that represent the functions of both general LSTM networks and specific components of the LSTM memory cell such as the input, forget, output, candidate gates, and other components. The library facilitates an intuitive simulation experience, allowing users to simply drag and drop the atomic models into DEVSimPy's workspace to construct and simulate complex LSTM models efficiently.

5.2. Proof-of-concept

To establish a proof of concept for our DEVS-based LSTM library, we began with a manual calculation of the outputs of an LSTM memory cell block and each one of its atomic components. This approach served as a reference to validate the correctness and reliability of the DEVS-based simulation results obtained using DEVSimPy.

The LSTM memory cell block was configured with specific input data ($c_t = 2$, $h_t = 1$, and $x_t = 1$), and predefined weights and biases for the gates as specified in Figure 4. This ensures a controlled environment for our calculations and subsequent simulations. We detail the manual calculations below, representing each LSTM gate and memory update:

$$\begin{aligned}
\text{Forget gate: } f_t &= \sigma(w_{fh} \times h_{t-1} + w_{fx} \times x_t + b_f) \\
&= \sigma(1 \times 2.7 + 1 \times 1.63 + 1.62) = \mathbf{0.997} \\
\text{Input gate: } i_t &= \sigma(w_{ih} \times h_{t-1} + w_{ix} \times x_t + b_i) \\
&= \sigma(1 \times 2 + 1 \times 1.65 + 0.62) = \mathbf{0.986} \\
\text{Candidate gate: } \tilde{c}_t &= \tanh(w_{ch} \times h_{t-1} + w_{cx} \times x_t + b_c) \\
&= \tanh(1 \times 1.41 + 1 \times 0.94 - 0.32) = \mathbf{0.97} \\
\text{Output gate: } o_t &= \sigma(w_{oh} \times h_{t-1} + w_{ox} \times x_t + b_o) \\
&= \sigma(1 \times 4.38 - 1 \times 0.19 + 0.59) = \mathbf{0.99} \\
\text{New Long-Term Memory: } c_t &= f_t \times c_{t-1} + i_t \times \tilde{c}_t \\
&= 0.997 \times 2 + 0.986 \times 0.967 = \mathbf{2.94} \\
\text{New Short-Term Memory: } h_t &= \tanh(c_t) \times o_t = \tanh(2.94) \times 0.99 \\
&= 0.994 \times 0.99 = \mathbf{0.98}
\end{aligned}$$

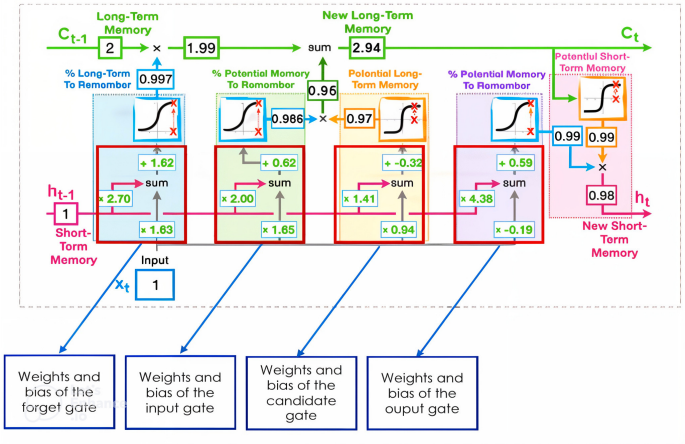

Figure 4. Detailed manual calculations for the LSTM cell proof of concept.

5.3. Application, results and discussion

Here, we detail the construction and simulation of the LSTM model within the DEVSimPy environment.

Model construction. As illustrated in Figure 5, the first step involves selecting the required atomic models from our DEVS-based library to assemble the LSTM memory cell coupled model (Figure 5). These atomic models represent the different functional aspects of the LSTM cell, such as the input, forget, candidate, and output gates. Each atomic model is interconnected to simulate the data flow and interactions within the LSTM architecture. We configure the initial parameters for each model, including input values $c_t = 1$, $h_t = 1$, and $x_t = 1$, as well as the weights and biases for the gates as depicted in Figures 6 and 7, where the parameters for each gate are set through the DEVSimPy's configuration panels.

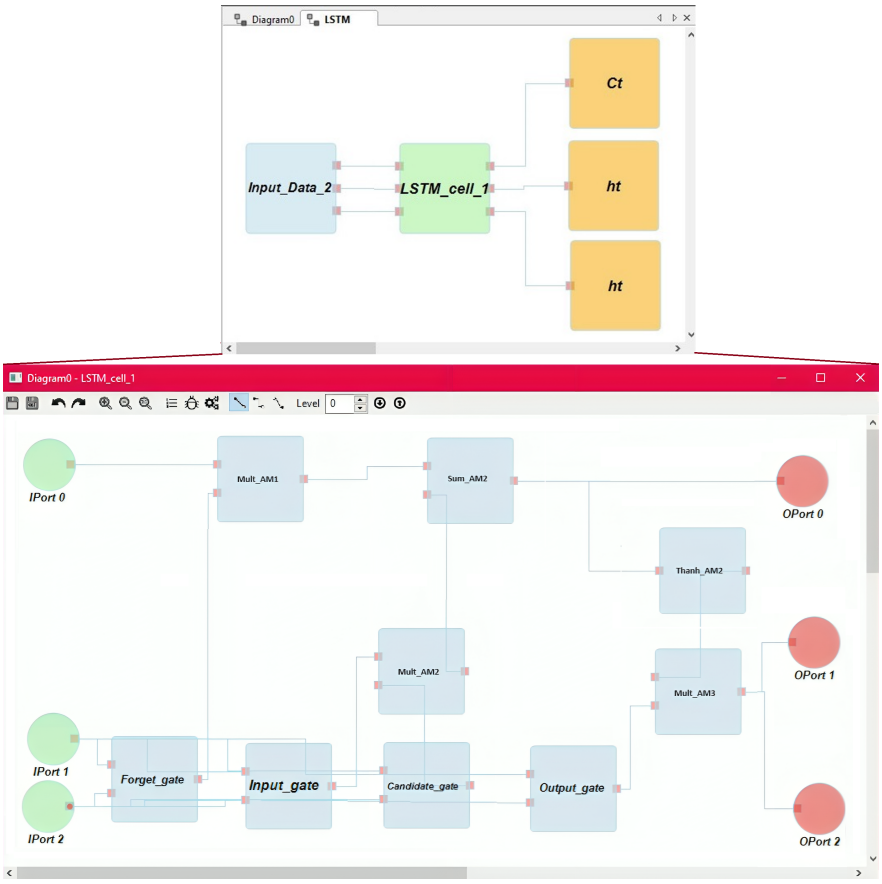

Figure 5. Coupled model of an LSTM memory cell, and its atomic components in DEVSimPy.

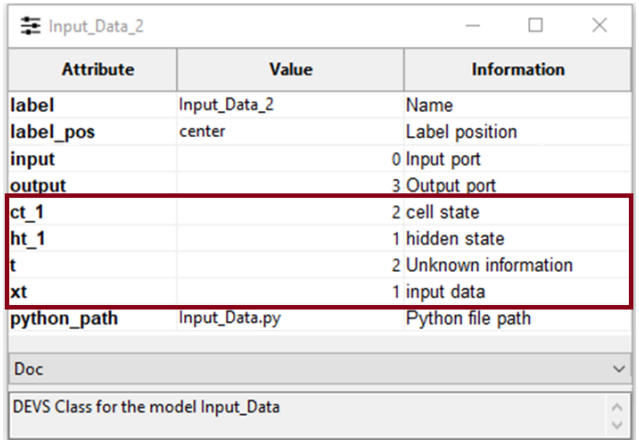

Figure 6. Set the properties of the `Input_Data` atomic model ($c_t = 1$, $h_t = 1$, and $x_t = 1$).

Simulation execution. Upon completing the model setup, the simulation is launched. The DEVSimPy environment provides various visualization plugins that display the initial states and state transitions occurring within each atomic model during the simulation. The visual tools help trace the data flow and understand the dynamic interactions among the LSTM components in real-time.

Results. Upon completion of the simulation, we meticulously analyzed the final outputs, specifically focusing on the new long-term and short-term memory values, c_t and h_t, which were recorded as 2.94 and 0.98 respectively. These results are in perfect alignment with our manual calculations, as detailed previously in Section 5.2. Illustrated in Figure 10, the correspondence of these values with those derived manually underscores the high fidelity of our DEVS-based LSTM simulation library. This consistency confirms the library's capability to accurately replicate expected outcomes and demonstrates its effectiveness in reproducing the dynamic behaviors and data transformations that are fundamental to the operation of LSTM networks. Additionally, as shown in Figure 9, our LSTM memory cell coupled model incorporates the *MessageCollector* atomic model for each gate to capture and log output values during the simulation. This setup allows us to observe the output values from each gate component directly, providing a granular view of the operational dynamics within the LSTM cell (Figure 10).

Discussion. The development and application of the DEVS-based LSTM library significantly enhance our understanding of LSTM networks, providing deep insights into how individual components interact and function cohesively

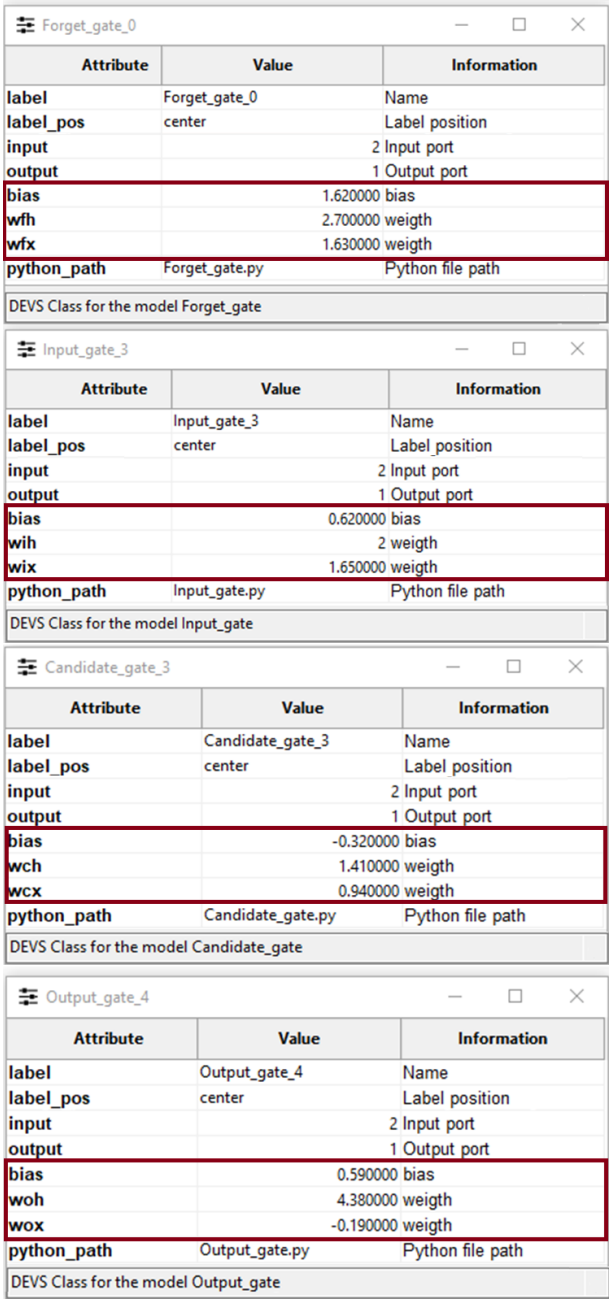

Figure 7. Configuration of the properties panel for the different gates in the LSTM coupled model.

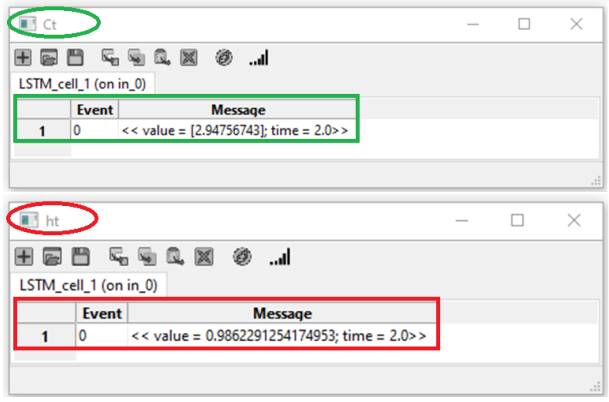

Figure 8. Simulation outputs of the LSTM memory cell in DEVSimPy.

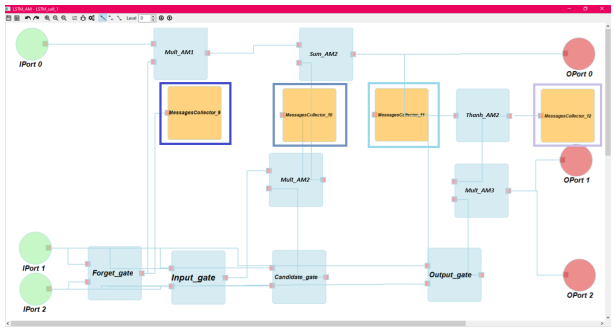

Figure 9. Integration of `Message_Collector` atomic models into the LSTM memory cell coupled model.

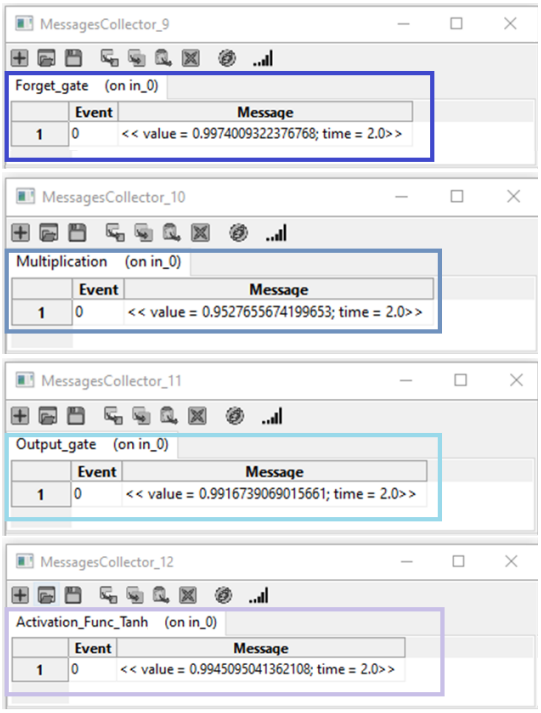

Figure 10. Outputs of the different `Message_Collector` atomic models corresponding to the different LSTM cell gates (Fig. 9).

within the network. By enabling modular design of ANNs, detailed visualization, and step-by-step simulation of data flows and operational dynamics, the library plays a crucial role in demystifying the complex mechanisms of LSTM operations. This approach clarifies the roles and effects of the different gates and states in LSTM cells and contributes to a broader understanding of ANNs behaviors in general. This work lays a foundational step towards a DEVS-based explainable approach for neural networks. By making the internal workings of LSTM networks transparent and observable, our DEVS-based LSTM library paves

the way for developing more interpretable and explainable models in machine learning. Such advancements are vital for increasing trustworthiness of ANN models in critical applications, promoting broader adoption and ethical use of XAI systems.

6. Conclusion and future work

In this paper, we introduced a novel DEVS-based approach for simulating LSTM neural networks within the DEVSimPy environment, enhancing the understanding of LSTM operations and contributing to explainable AI (XAI). This approach, realized through a DEVS-based library, provides a modular framework for detailed simulation of LSTM functions. A proof-of-concept case study confirmed the library's ability to transparently replicate LSTM behaviors and data transformations, essential for elucidating the internal processes of LSTM networks.

Future work will involve applying the DEVS-based LSTM library to more complex scenarios with real data and larger-scale models to assess its robustness and scalability. We also plan to broaden the library's scope to include other types of Recurrent Neural Networks (RNNs), thereby increasing its utility and adaptability for diverse neural network architectures.

7. Funding

This work was funded by the Computer Science & Systems Laboratory (LIS) of the University of Aix-Marseille, and partially supported by the Engineering science, computer science and imaging laboratory (ICube) of the University of Strasbourg.

References

Adelani, D. I. and Traoré, M. K. (2016). Enhancing the reusability and interoperability of artificial neural networks with devs modeling and simulation. *International Journal of Modeling, Simulation, and Scientific Computing*, 7(02):1650005.

Alshareef, A., Blas, M. J., Bonaventura, M., Paris, T., Yacoub, A., and Zeigler, B. P. (2022). Using devs for full life cycle model-based system engineering in complex network design. In *Advances in Computing, Informatics, Networking and Cybersecurity: A Book Honoring Professor Mohammad S. Obaidat's Significant Scientific Contributions*, pages 215–266. Springer.

Anbil Parthipan, S. C. (2020). On challenges in training recurrent neural networks.

Buhrmester, V., Münch, D., and Arens, M. (2021). Analysis of explainers of black box deep neural networks for computer vision: A survey. *Machine Learning and Knowledge Extraction*, 3(4):966–989.

Capocchi, L., Santucci, J. F., Poggi, B., and Nicolai, C. (2011). Devsimpy: A collaborative python software for modeling and simulation of devs systems. In *2011 IEEE 20th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 170–175. IEEE.

Islam, M. U., Mozaharul Mottalib, M., Hassan, M., Alam, Z. I., Zobaed, S., and Fazle Rabby, M. (2022). The past, present, and prospective future of xai: A comprehensive review. *Explainable Artificial Intelligence for Cyber Security: Next Generation Artificial Intelligence*, pages 1–29.

Kim, T. G. and Zeigler, B. P. (1987). The devs formalism: hierarchical, modular systems specification in an object oriented framework. Technical report, Institute of Electrical and Electronics Engineers (IEEE).

Kisvari, A., Lin, Z., and Liu, X. (2021). Wind power forecasting–a data-driven method along with gated recurrent neural network. *Renewable Energy*, 163:1895–1909.

Louis, F. (2024). Long short-term memory (lstm) networks.

Roscher, R., Bohn, B., Duarte, M. F., and Garcke, J. (2020). Explainable machine learning for scientific insights and discoveries. *Ieee Access*, 8:42200–42216.

Saeed, W. and Omlin, C. (2023). Explainable ai (xai): A systematic meta-survey of current challenges and future opportunities. *Knowledge-Based Systems*, 263:110273.

Samek, W., Wiegand, T., and Müller, K.-R. (2017). Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296*.

Saraswat, D., Bhattacharya, P., Verma, A., Prasad, V. K., Tanwar, S., Sharma, G., Bokoro, P. N., and Sharma, R. (2022). Explainable ai for healthcare 5.0: opportunities and challenges. *IEEE Access*, 10:84486–84517.

Sehili, S., Capocchi, L., Santucci, J. F., Lavirotte, S., and Tigli, J.-Y. (2015). Discrete event modeling and simulation for iot efficient design combining wcomp and devsimpy framework. In *5th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*.

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

The-RED-Network (2024). Devs tools. https://devs-network.org/tools/. Accessed: May 2024.

Tiddi, I. and Schlobach, S. (2022). Knowledge graphs as tools for explainable machine learning: A survey. *Artificial Intelligence*, 302:103627.

Toma, S., Capocchi, L., and Federici, D. (2011). A new devs-based generic artificial neural network modeling approach. *Proceedings of the EMSS*.

Torres, J. F., Hadjout, D., Sebaa, A., Martínez-Álvarez, F., and Troncoso, A. (2021). Deep learning for time series forecasting: a survey. *Big Data*, 9(1):3–21.

Van Houdt, G., Mosquera, C., and Nápoles, G. (2020). A review on the long short-term memory model. *Artificial Intelligence Review*, 53(8):5929–5955.