Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

pymodconn: A python package for developing modular sequence-to-sequence control-oriented deep neural networks

Gaurav Chaudhary [a,*], Hicham Johra [b], Laurent Georges [a], Bjørn Austbø [a]

[a] Department of Energy and Process Engineering, NTNU, Trondheim, Norway
[b] Department of the Built Environment, Aalborg University, Aalborg Øst, Denmark

ARTICLE INFO

Keywords:
Control-oriented deep neural networks
System dynamics prediction
Deep neural networks

ABSTRACT

This paper introduces "*pymodconn*", a comprehensive python package developed for constructing modular sequence-to-sequence control-oriented deep neural networks. These deep neural networks (DNNs) are designed to predict the future dynamics of complex time-dependent systems for given known future data, e.g., control inputs, using past known system dynamics and control inputs. The strength of DNNs in modeling complex systems is well known, but developing an optimal deep learning-based model can be a resource-intensive task. This package streamlines this process, simplifying model architecture selection and fine-tuning. The key strength of *pymodconn* lies in its high-level modularity, enabling users to design their DNN architectures in a flexible manner via a simple text-based configuration file. This flexibility and the comprehensive nature of *pymodconn* considerably reduce the development efforts and time for applications where precise control over system dynamics is necessary.

Code Metadata

C1	Current code version	V2.0.0
C2	Permanent link to code/ repository used for this code version	https://github.com/gaurav306/pymodconn
C3	Permanent link to reproducible capsule	https://zenodo.org/record/8150987
C4	Legal code license	MIT
C5	Code versioning system used	GIT
C6	Software code languages, tools and services used	Python >= 3.9.16
C7	Compilation requirements, operating environments and dependencies	Python>=3.9.16 and packages: jsonschema==4.17.3, numpy==1.24.3, PyYAML==6.0, ruamel.base==1.0.0, tensorflow==2.12.0
C8	If available, link to developer documentation/manual	https://github.com/gaurav306/pymodconn/blob/master/README.md
C9	Support email for questions	gaurav.chaudhary@ntnu.no

Software Metadata

S1	Current software version	V2.0.0
S2	Permanent link executables of this version	https://github.com/gaurav306/pymodconn
S3	Permanent link to reproducible capsule	https://zenodo.org/record/8150987
S4	Legal software license	MIT
S6	Computing platforms/ Operating Systems	Linux, Windows, macOS
S7	Installation requirements & dependencies	Python>=3.9.16 and packages: jsonschema==4.17.3, numpy==1.24.3, PyYAML==6.0, ruamel.base==1.0.0, tensorflow==2.12.0
S8	If available, link to developer documentation/manual	https://github.com/gaurav306/pymodconn/blob/master/README.md
S9	Support email for questions	gaurav.chaudhary@ntnu.no

1. Motivation and significance

A dynamic system is a system in which state variables evolve over time based on a fixed set of rules and interactions, often described by differential equations. As dynamic systems grow in complexity, effective management and control become imperative [1]. Predictive strategies, especially model predictive control, have been recognized for their

* Corresponding author.
E-mail address: gaurav.chaudhary@ntnu.no (G. Chaudhary).

https://doi.org/10.1016/j.softx.2023.101599
Received 15 July 2023; Received in revised form 23 October 2023; Accepted 27 November 2023
Available online 7 December 2023

potency in reducing operational costs while accommodating fluctuating boundary conditions and internal dynamics [2,3]. Central to the deployment of such advanced predictive control methodologies is the formulation of a control-oriented dynamic model that can predict how different control signals influence the system's behavior and outcomes.

Like all models, a control-oriented dynamic model can be broadly categorized into three types: white-box, grey-box, and black-box models [2]. White-box models, also known as physics-based models, require detailed system information. They are grounded in first-principle equations to calculate the behavior and response of the system to different inputs. Some recent examples of white-box models are CoBo [4] for control-oriented building dynamics simulation, WFSim [5] for control-oriented wind farm simulation, and control-oriented models for chemical batteries [6,7]. Grey-box or reduced-order models find a middle ground by using simplified dynamic equations combined with datasets for parameter identification. A common method used in this category is resistance-capacitance (RC) networks, which represent system elements and energy flows in a simplified manner. Examples of that include use of RC control-oriented models for building control [8,9] and use reduced order models for lithium-ion battery control [10]. Black-box models, sometimes referred to as data-driven models, predominantly leverage statistical or machine learning techniques to predict system responses [11]. Examples like AutoRegressive Integrated Moving Average (ARIMA) based control-oriented models for electric vehicle energy management [12], Support Vector Machines (SVMs) based control-oriented models for diesel engine emission control [13], and Deep Neural Networks (DNNs) based control-oriented models for building control [14] and control-oriented flow estimation [15] fall under this category.

With the proliferation of data from sensors and monitoring technologies, DNNs in general have emerged as a powerful tool for predicting system dynamics, proving particularly effective for complex systems. The primary advantages of DNNs include their capacity to learn from high-dimensional data and their ability to capture non-linear relationships [16–18]. It has been shown that a DNN with enough hidden units can approximate any arbitrary continuous function defined on a closed and bounded set [19,20]. Given their capability to process and learn from vast and intricate data sets, Control-Oriented deep Neural Networks (CONNs) can be tailored to formulate control-oriented dynamic models that are both accurate and fast.

A CONN, as illustrated in Fig. 1, is a deep neural network structured to predict the future behavior of complex dynamic systems, using both *Known Past* and *Known Future data. Known Past data* includes historical metrics like weather, system behavior, and control inputs, all used to train the CONN to discern the correlations between these variables and their influence on the system. *Known Future data* comprises forecasted weather data, schedules and planned control inputs, which represent the anticipated conditions and desired actions the system will face in the future. The CONN uses this data to predict *Unknown Future data*, meaning system behavior or states that are not yet known. This prediction enables the control system to estimate the influence of the control input and adapt its actions to improve performance.

Despite the strength of DNNs in modeling complex systems, selecting a suitable model architecture and fine-tuning the model can be cumbersome. Hence improving the process of developing such architecture is highly desirable. **The primary research objective of this work is to devise a comprehensive tool or framework that facilitates streamlined creation of modular sequence-to-sequence control-oriented deep neural networks, specifically tailored for dynamic system predictions.** Addressing this objective, the tool "*pymodconn*" is introduced. It provides a highly modular and user-friendly approach to creating CONNs by using a text-based configuration file as input from users.

While there are various modular DNN development packages, there are some gaps that *pymodconn* aims to address:

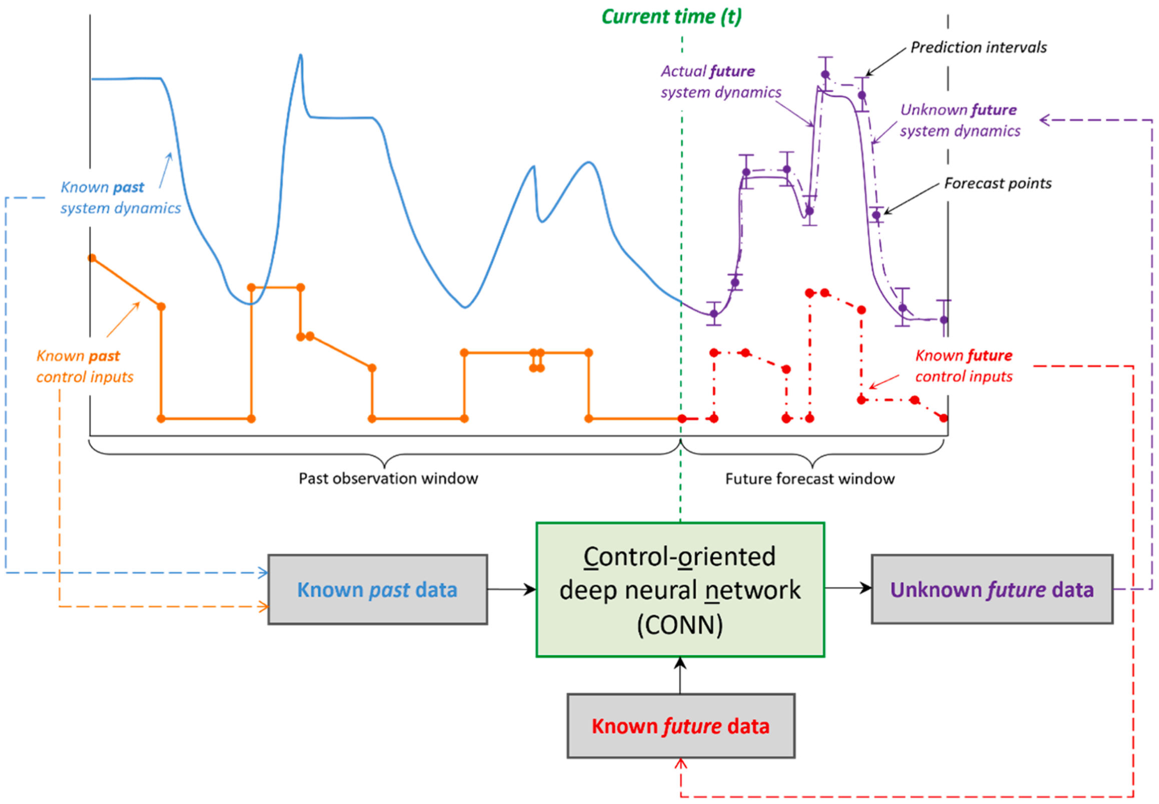

Current time (t)
Prediction intervals
Actual future system dynamics
Unknown future system dynamics
Known past system dynamics
Forecast points
Known past control inputs
Known future control inputs
Past observation window
Future forecast window
Known past data
Control-oriented deep neural network (CONN)
Unknown future data
Known future data

Fig. 1. Abstract depiction of a control-oriented deep neural network (CONN). The CONN uses known past data and planned future control inputs to predict the future behavior of a dynamic system, shown here with a hypothetical system and control inputs.

- **Limitations of Existing Libraries**: Several deep learning libraries such as AutoKeras [21], KerasCV [22], DeepCTR [23], Caffe [24], FastAI [25], and DeepChem [26] have emerged to facilitate the modular development of deep neural networks. However, they present certain limitations:
  - Ø **Field Specificity:** These libraries, while powerful, are designed for other specific application domains. For example:
    - ■ **AutoKeras** [21] is designed to simplify the machine-learning model-design process but lacks an extensive focus on dynamic systems prediction.
    - ■ **KerasCV** [22] is tailored for machine vision tasks.
    - ■ **DeepCTR** [23] is dedicated to deep-learning based click-through rate prediction models.
    - ■ **Caffe** [24] is notably geared towards convolutional neural networks, for computer vision tasks.
    - ■ **FastAI** [25] aims to make deep learning more user-friendly but is built atop the PyTorch library
    - ■ **DeepChem** [26] is dedicated to applications in drug discovery, quantum chemistry and biology.
  - Ø **Inclination towards Larger Models:** Many of the above-mentioned libraries tend to focus on large and complex models only. Even though these models are powerful, they are not always the most practical. *Pymodconn*'s modularity provides capability to adjust depth.
- **Open-source nature:** Many of the truly modular DNN development tools remain inaccessible to the wider community. Both research professionals and corporations often develop these tools for in-house testing and development, keeping their advanced features under wraps. In contrast, *pymodconn* tries to break this trend. To the best of the authors' knowledge, it is the first open-source package providing such comprehensive modular model development capabilities.
- **User-friendly configuration**: *pymodconn*'s design separates configuration from code, which improves research flexibility and efficiency. This design allows for batch predictions, GPU task distribution, and enhanced result reproducibility contributing to the principles of open science. It also supports comprehensive ablation analyses, fostering a better understanding of time series prediction techniques for control systems.

2. Software description

The *pymodconn* package is built upon the robust Keras library [27], utilizing its user-friendly, modular, and efficient structures for developing and training deep learning models. The use of Keras facilitates integration with other machine learning tools, allowing users to create customized pipelines for time series prediction, tailored to their specific needs. It is based on a sequence-to-sequence architecture [28] and integrates a broad spectrum of established time series prediction mechanisms.

2.1. Software architecture

As outlined in the introduction, in the context of the CONN architecture, two inputs are essential: the *Known past data* (e.g., known past system dynamics + past control inputs) and the *Known future data* (e.g., targeted control inputs + other known future data). The network's objective is to generate an output representing the *Unknown future data* (e.g., future system dynamics). The following section briefly describes different components of CONN, how they are connected to each other, and how they can improve prediction accuracy. Pseudocode for the components described here is given in Appendix B.

For clarity, all the user-configurable parameters in the configuration file are highlighted in red and indicated by dashed lines in the subsequent figures related to the model architecture.

2.1.1. Top-level architecture

As illustrated in Fig. 2, the CONN's basic structure consists of an Encoder block processing the *Known past data*, and a *Decoder* block handling the *Known future data*. The *Encoder* block processes the *Known past data*, generating two distinct outputs: a sequence of hidden states, also called encoder hidden states, and the final encoder state. The encoder hidden states encapsulate the temporal structure inherent in the past data, while the final encoder state provides a summarized representation of this data. Each *Decoder* block is designed to receive and process the outputs from the *Encoder* block. The encoder hidden states serve as a temporal context for each *Decoder* block, while the final encoder state is used to initialize the state of the *Decoder* block.

The outputs from the *Decoder* blocks are subsequently passed through an *Output type* block. This block refines the output, either in the form of a point forecast or as a probabilistic (interval) forecast. For point forecasts, the *Output type* block simply passes the output of the Decoder forward without modification. However, for probabilistic forecasts, the *Output type* block introduces additional layers to facilitate the generation of interval forecasts, providing a measure of uncertainty for the predictions. The users can choose between two distinct methodologies for probabilistic forecasts, both adapted from the DeepTCN model architecture [29]. In the configuration file, these are activated by the terms *'nonparametric'* and *'parametric'*, respectively. Pseudocode for the top-level model generation is shown in Fig. B.1 in Appendix B.

2.1.2. Encoder block

The *Encoder* block is a key building block in the CONN architecture (see Fig. 2). It processes the *Known past data*, which includes the known past system states and past control inputs. The outputs of this block are essential for setting up the context for the subsequent *Decoder* blocks, enabling them to make informed predictions about the *Unknown future system states*.

In the *Encoder* block, the input first encounters a *Dense* layer, which boosts data dimensions to capture intricate patterns. The *Dense* layer's broad interconnectedness allows for diverse feature learning, optimizing model performance. A subsequent *Dropout* layer curbs overfitting by randomly nullifying a portion of weights during each training update. The users can then select among three established time series prediction architectures: Temporal Convolutional Networks (TCN) [29,30], Recurrent Neural Network (RNN) [31–33], and Self Multi-Head Attention (Self-MHA) mechanisms [34,35]. These can be employed individually, in combination or altogether, offering flexible configuration for ablation analyses. Inclusion of the *RNN* block in the *Encoder* generates encoder hidden states, representing the temporal structure of the data and providing essential temporal context for the *RNN* blocks in the *Decoder*. Ultimately, the *Encoder* block yields a condensed, feature-rich past data representation, preparing the terrain for the *Decoder* block to accurately project future system dynamics. Details of the *MHA* block and the *RNN* block are given in subsections 2.1.6 and 2.1.7, respectively. Pseudocode for the *Encoder* generation is shown in Fig. B.2 in Appendix B.

2.1.3. Decoder block

The *Decoder* block serves as the bridge between the processed past data and the future predictions (see Fig. 2). It takes the outputs of the *Encoder* block, i.e., the sequence of encoded states and the final encoder state, along with the *Known future data*, and estimates the preliminary future system dynamics. This crucial component leverages the context provided by the *Encoder* block to ensure that the predictions are informed by the temporal patterns identified in the past data. Details of different blocks inside the *Decoder* block are described later in this

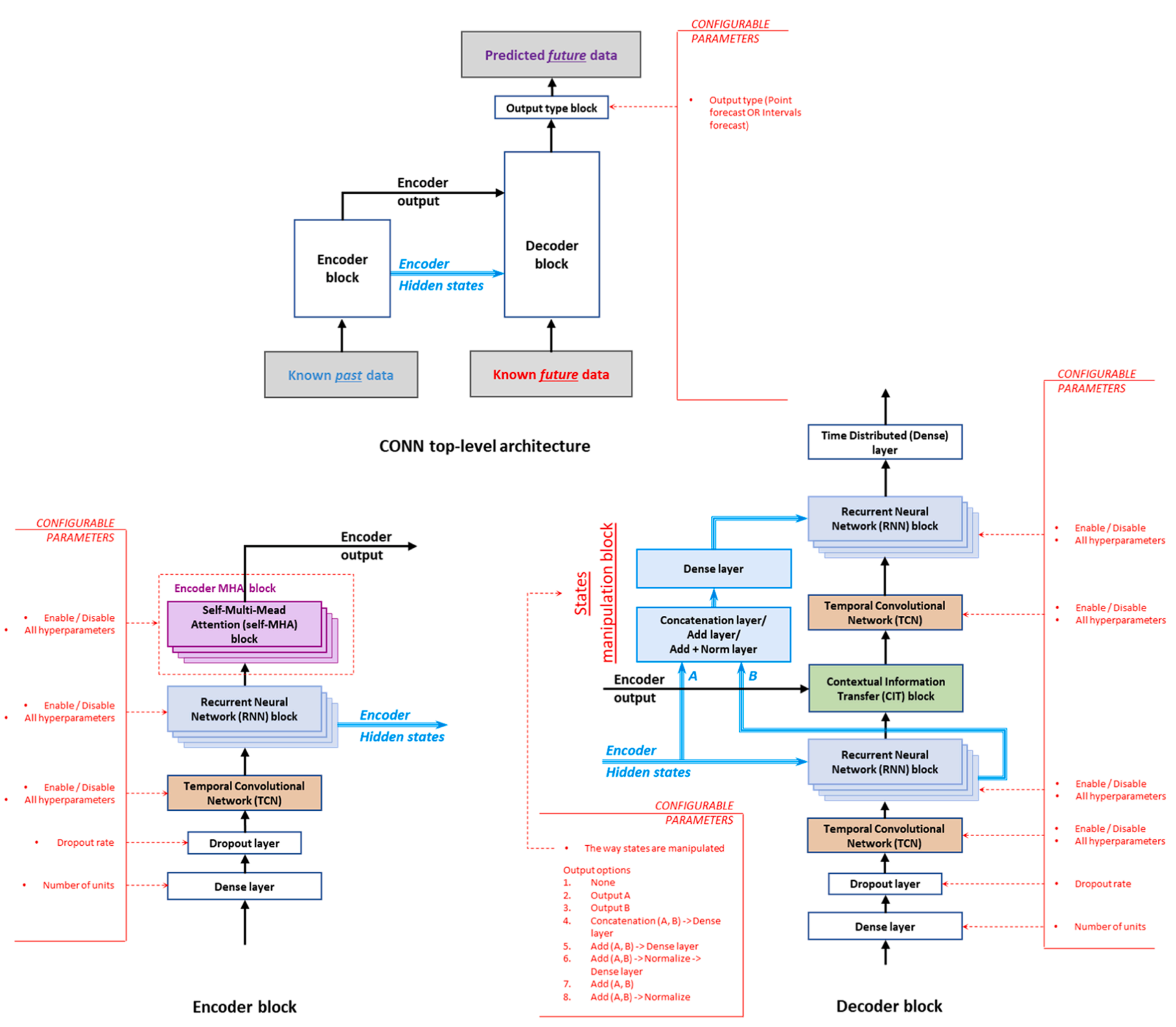

CONFIGURABLE PARAMETERS
Predicted future data
Output type block
Output type (Point forecast OR Intervals forecast)
Encoder output
Encoder block
Encoder Hidden states
Decoder block
Known past data
Known future data
CONN top-level architecture
CONFIGURABLE PARAMETERS
Encoder output
Encoder MHA block
Self-Multi-Mead Attention (self-MHA) block
Enable / Disable
All hyperparameters
Recurrent Neural Network (RNN) block
Encoder Hidden states
Temporal Convolutional Network (TCN)
Dropout rate
Dropout layer
Number of units
Dense layer
Encoder block
States manipulation block
Dense layer
Concatenation layer/ Add layer/ Add + Norm layer
A
B
Encoder output
Encoder Hidden states
CONFIGURABLE PARAMETERS
The way states are manipulated
Output options
1. None
2. Output A
3. Output B
4. Concatenation (A, B) -> Dense layer
5. Add (A, B) -> Dense layer
6. Add (A,B) -> Normalize -> Dense layer
7. Add (A, B)
8. Add (A,B) -> Normalize
Time Distributed (Dense) layer
Recurrent Neural Network (RNN) block
Temporal Convolutional Network (TCN)
Contextual Information Transfer (CIT) block
Recurrent Neural Network (RNN) block
Temporal Convolutional Network (TCN)
Dropout layer
Dense layer
CONFIGURABLE PARAMETERS
Enable / Disable
All hyperparameters
Dropout rate
Number of units
Decoder block

Fig. 2. The top-level architecture of the CONN, *Encoder* block, *Decoder* block and *States Manipulation* block within the CONN framework.

section.

The *Decoder* block follows a structure similar to the *Encoder* block. It initiates the processing of its input with a *Dense* layer, followed by a *Dropout* layer. After these initial transformations, the processed input proceeds to either a *TCN* block or an *RNN* block, or both, depending on the users' configuration. Notably, the *RNN* block in the *Decoder* uses the encoder hidden states, as its initial states. Details of the *CIT* block are given in subSections 2.1.4. Pseudocode for the *Decoder* generation is shown in Fig. B.3 in Appendix B.

2.1.4. *Contextual information transfer block*

Upon exiting the *RNN* block (or alternatively the *TCN* block, or directly as input to the *Decoder* block if both *RNN* and *TCN* blocks are disabled), the data flow enters the *Contextual Information Transfer* (CIT) block. The *CIT* block acts as a bridge between the *Encoder* and *Decoder*, facilitating effective past-to-future data information transfer for precise predictions. Three distinct configurations exist for the *CIT* block, as shown in Fig. 3. Option 1 uses a straightforward approach, reshaping *Encoder* output to align with future prediction dimensions in a *Reshape* layer. The reshaped output is combined with output from the *RNN* and/ or *TCN* block, and then processed by a *Dense* layer. However, this option does not consider the sequential nature of the data. Option 2 uses an *Attention* layer instead of a *Reshape* layer. The attention mechanism accentuates key input elements, proving beneficial when handling dynamic inputs with abrupt changes. Users can opt for Bahdanau (additive) [34] or Luong (scaled dot-product) attention [35]. Option 3 incorporates Transformer elements, featuring self-MHA and cross-MHA attention mechanisms. Self-MHA uncovers key temporal patterns in Known future data, while cross-MHA integrates self-MHA and encoded states outputs, offering temporal context from past data when processing Known future data. A subsequent *Gated Residual Network* block (GRN) [36] follows cross-MHA.

CIT blocks can be stacked multiple times, promoting the learning of complex data representations. Stacking facilitates hierarchical learning, with simpler patterns recognized in lower layers and complex structures understood in higher ones. Pseudocode for the *CIT* block generation is shown in Fig. B.4 in Appendix B.

2.1.5. *States manipulation block*

In the later stage of the *Decoder* block, the output from the *CIT* block is passed through a combination of *TCN* and/or *RNN* blocks. The memory retention feature of RNNs is used to recall earlier parts of the sequence, assisting in accurate predictions. The initial states of this RNN come from a mix of hidden states from the *RNN* blocks in the *Encoder* and *Decoder* blocks, manipulated in the *States Manipulation* block (see Fig. 2). Users can manipulate these states in various ways, including outputting

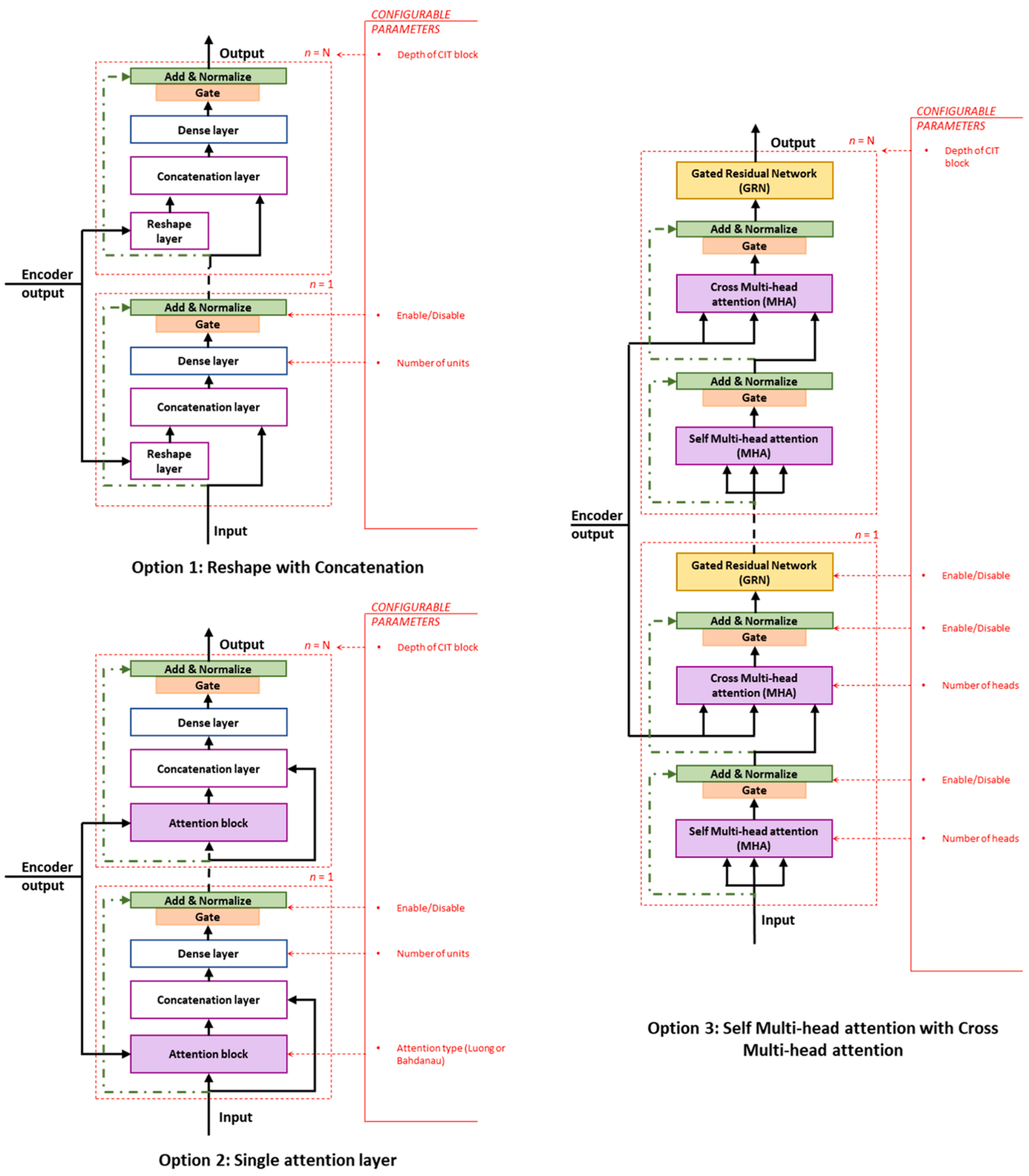

CONFIGURABLE PARAMETERS
Output
n = N
Depth of CIT block
Add & Normalize
Gate
Dense layer
Concatenation layer
Reshape layer
Encoder output
n = 1
Enable/Disable
Number of units
Input
Option 1: Reshape with Concatenation
Attention block
Attention type (Luong or Bahdanau)
Option 2: Single attention layer
Gated Residual Network (GRN)
Cross Multi-head attention (MHA)
Self Multi-head attention (MHA)
Number of heads
Option 3: Self Multi-head attention with Cross Multi-head attention

Fig. 3. The three architectural options for the *CIT* block within the CONN model.

hidden states from either of the two *RNN* blocks, or processing them through combinations of *Concatenation, Addition, Normalization* and *Dense* layers. Concatenation expands the input dimensionality for the subsequent *Dense* layer, enhancing the learning of complex representations. Addition merges the information from both states, offering a more comprehensive representation. Normalization, in some cases, stabilizes the deep learning process by ensuring stable hidden states distribution.

The *Decoder* block ends with a *Time Distributed Dense* layer that reshapes the data to align with the dimensions of the future prediction window. The Encoder *Multi-Head Attention* block (as shown in Fig. 2), *Recurrent Neural Network* block (as shown in Fig. 2), *Gate Resdiual Network* block (as shown in Fig. 3) and *Add&Normalize* + *Gate* block (as shown in Fig. 3) are further described in Appendix A.

3. Illustrative examples

There are two methods available for users to utilize *pymodconn*. The first is to clone or download the project directly from the GitHub repository. The second method is to install it via pip, Python's package installer, by executing the following command: ***"pip install pymodconn"***. Its v2.0.0 has also been deposited on Zenodo.org as a permanent copy [37].

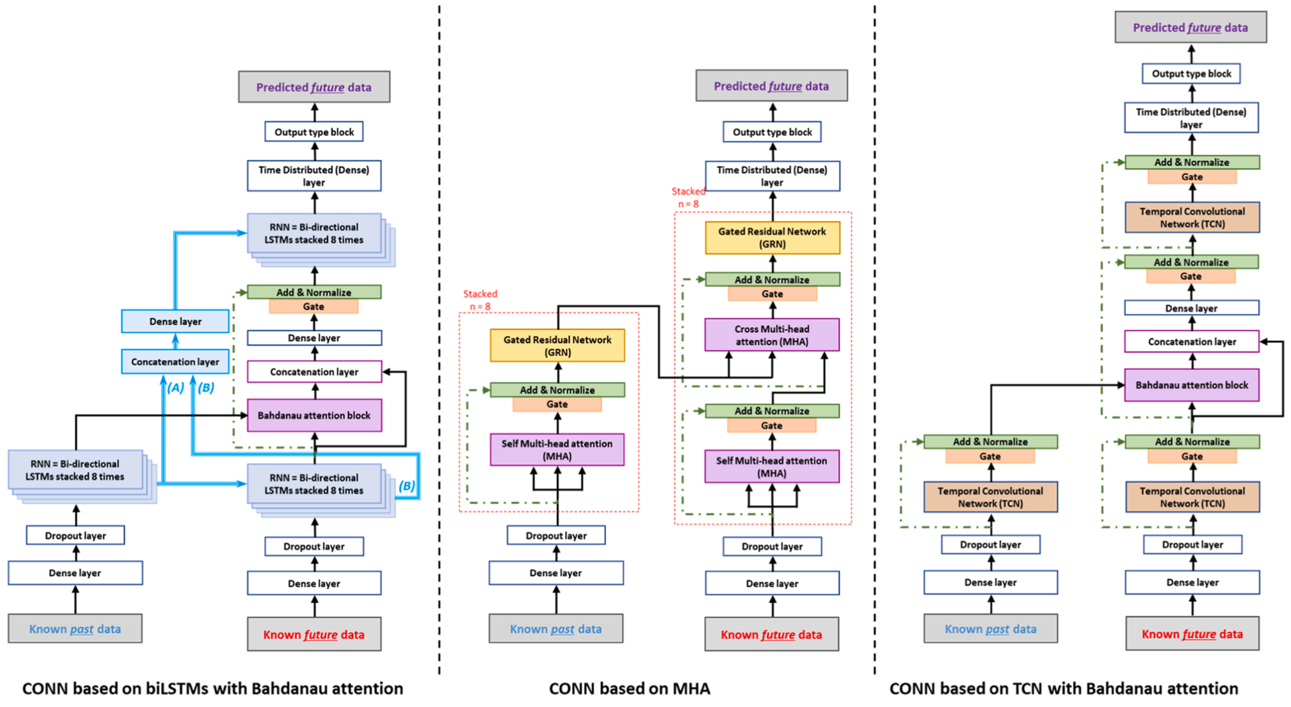

Predicted future data
Output type block
Time Distributed (Dense) layer
RNN = Bi-directional LSTMs stacked 8 times
Add & Normalize
Gate
Dense layer
Concatenation layer
Bahdanau attention block
(A)
(B)
Dropout layer
Known past data
Known future data
Stacked n = 8
Gated Residual Network (GRN)
Self Multi-head attention (MHA)
Cross Multi-head attention (MHA)
Temporal Convolutional Network (TCN)
CONN based on biLSTMs with Bahdanau attention
CONN based on MHA
CONN based on TCN with Bahdanau attention

Fig. 4. Schematics of example models constructed using *pymodconn*.

Examples on how to implement the package are given in Appendix C. Fig. 4 shows the schematics of three example models exemplifying the modularity and flexibility of *pymodconn*. Additional documentation can be found in the *README.md* file within the GitHub repository.

4. Impact

Pymodconn lowers the technical hurdles to implement cutting-edge DNNs, fostering quicker innovation and enhanced productivity. Its capability to facilitate rapid development and testing makes it an indispensable learning resource for students and professionals delving into DNNs for system dynamics and control. With its user-friendly approach to crafting and deploying neural networks, *pymodconn* stands to significantly elevate everyday practices.

This package was recently used by authors to develop a prediction model based on Multi-head Attention (MHA) and Transformer components to predict indoor air temperature evolution with heating setpoint, opening and closing of windows as control inputs for an office building [38].

Unlike existing libraries, *pymodconn* offers pre-configured architectures specifically designed for CONN models. Working with other solutions entails a steep learning curve to understand the assembly of components/layers for control-oriented tasks, along with the manual coding of configurations. This often raises concerns about coding errors and the burden of maintaining multiple codebase versions, thereby extending the development cycle. In contrast, pymodconn facilitates quicker iterations, and a more agile process.

5. Conclusion

In this paper, the Python package *pymodconn* has been presented as a comprehensive tool for constructing sequence-to-sequence control-oriented deep neural networks. This unique package offers users the ability to design deep neural networks with a broad range of high-performance time series prediction architectures. The distinctiveness of *pymodconn* lies in its high-level modularity, facilitated by the separation of configuration from code. This approach allows users to adjust architectures and parameters through a user-friendly text-based configuration file.

Despite its modularity and flexibility, there are some limitations in *pymodconn*. The platform is currently tailored exclusively for TensorFlow and Keras, though future updates are planned to incorporate PyTorch support. Additionally, its unique approach of decoupling configuration from the code can present a daunting learning curve, especially for newcomers who might find the multitude of configuration options difficult to navigate. There is also a notable absence of built-in tools for hyperparameter tuning/model optimization, necessitating the reliance on external tools or manual effort. Moreover, in situations requiring advanced designs or the introduction of custom layers, users might find themselves delving into the package's source code for adjustments. Despite these intricacies, the openness and adaptability of *pymodconn* create new opportunities for future exploration and innovation within the field of control-oriented deep neural networks.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgements

The authors acknowledge the support from the strategic research program ENERSENSE at Norwegian University of Science and Technology (NTNU).

Appendix A: Additional components of *pymodconn*

Fig. A.1, Fig. A.2

The following section provides information about additional components of *pymodconn, in addition* to those presented in Section 2.

A.1 Encoder Multi-head attention block

The Encoder *MHA* block is also inspired by the Transformer model [39]. The first layer, i.e., a self-MHA layer enables the model to discern and focus on key temporal patterns within the known past data, effectively creating a summarized representation of important trends and dependencies. After this, the data undergoes transformation by a *GRN* block. The output of the self-MHA can optionally be connected back to the original input of the Encoder *MHA* block via a gated *Add* + *Normalize* block, providing a form of residual connection that can help stabilize learning. The Encoder *MHA* block can also be stacked, providing depth and increasing the model's capacity to learn intricate patterns in the data. The number of blocks is given by the users, further highlighting the flexibility of the CONN architecture (see Fig. A.1). Pseudocode for the *MHA* block generation is shown in Fig. B.5 in Appendix B.

A.2 Recurrent Neural Network block

The *RNN* block exploits RNN architectures' unique capacities for processing sequential data and retaining past inputs. Users can choose the depth of this block depth and choose between SimpleRNN [31], Long Short-Term Memory (LSTM) [32], or Gated Recurrent Unit (GRU) [33] architectures (see Fig. A.1), with the added choice of bidirectional RNNs to capture overlooked cyclic patterns, providing a more comprehensive understanding of recurrent patterns. Each layer includes a residual connection for stable gradient propagation during training. An additional *GRN* block transforms the data after the RNNs, maintaining the stability of the model. Pseudocode for the *RNN* block generation is shown in Fig. B.6 in Appendix B.

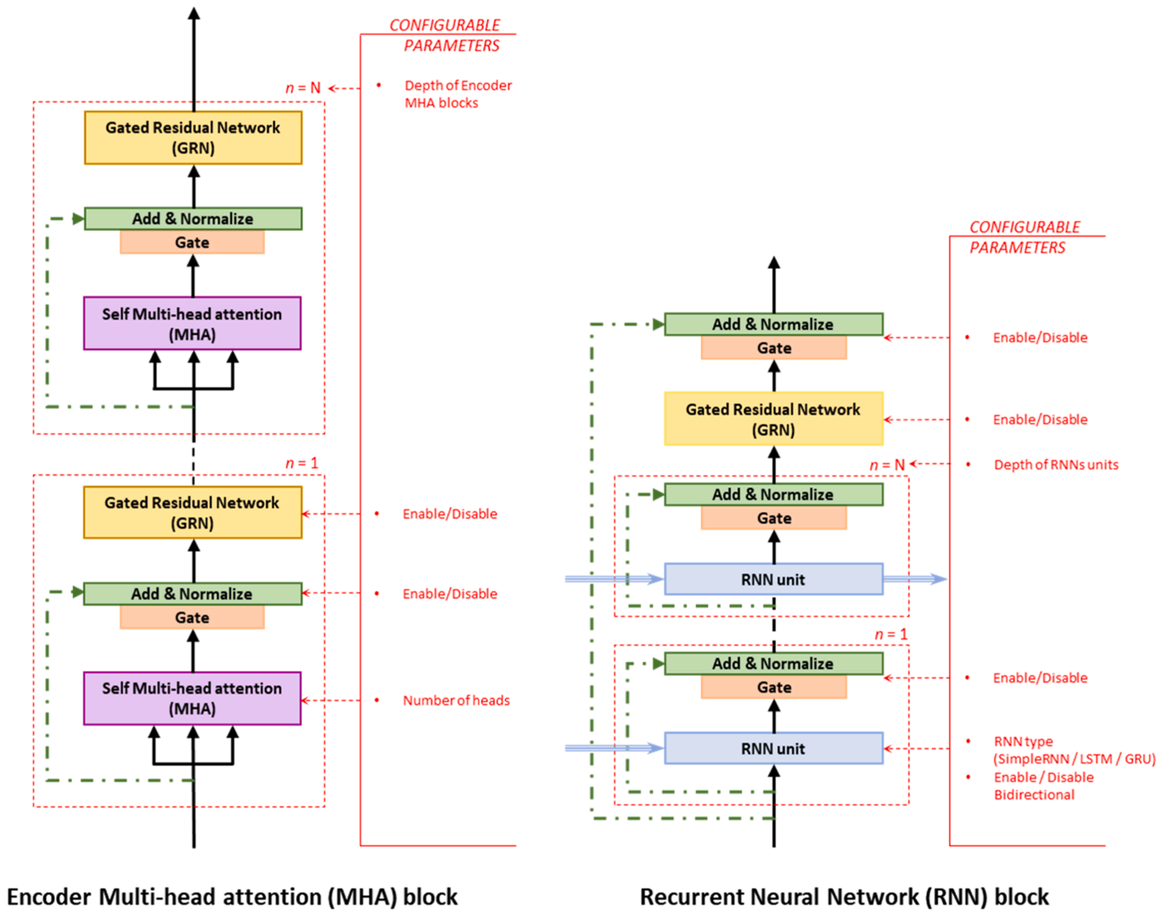

CONFIGURABLE PARAMETERS
n = N
Depth of Encoder MHA blocks
Gated Residual Network (GRN)
Add & Normalize
Gate
Self Multi-head attention (MHA)
n = 1
Enable/Disable
Enable/Disable
Number of heads
Encoder Multi-head attention (MHA) block
CONFIGURABLE PARAMETERS
Enable/Disable
Enable/Disable
Depth of RNNs units
RNN unit
Enable/Disable
RNN type (SimpleRNN / LSTM / GRU)
Enable / Disable Bidirectional
Recurrent Neural Network (RNN) block

Fig. A.1. Detailed schematic for the Encoder *MHA* block and the *RNN* block in the CONN architecture.

A.3 Add & Normalize + Gate block

The *CIT* block, across all configurations, includes residual connections through gated *Add & Normalize* blocks [40] (see Fig. A.2). These blocks aid in managing information flow and stabilizing learning. Residual connections in the *Add* layer help counter the vanishing gradients issue common in deep networks, enabling more robust learning by adjusting predictions based on the unaltered input. The *Normalize* layer applies layer normalization, standardizing inputs to each layer as per training sample.

A gating mechanism, more specifically a Gated Linear Unit (GLU), is used as an activation function. The GLU was introduced in the Temporal

Fusion Transformers (TFT) model [36] as a way to control the flow of information through the network. The GLU enables selective emphasis on certain data features. Components of GLU determines the extent to which each feature should be 'allowed through the gate', offering flexibility and adaptability in focusing on the most relevant features, potentially improving predictive performance.

A.4 Gated Residual Network

The *GRN* block within the CONN architecture masterfully manages multiple varying inputs. It employs GLU for selective information retention [36] (see Fig. A.2). This is particularly useful in the CONN context, where diverse inputs such as past system states and future control inputs are processed. The *GRN* block also integrates a gating strategy with a residual connection and *Dense* layer outputs, promoting stable model training.

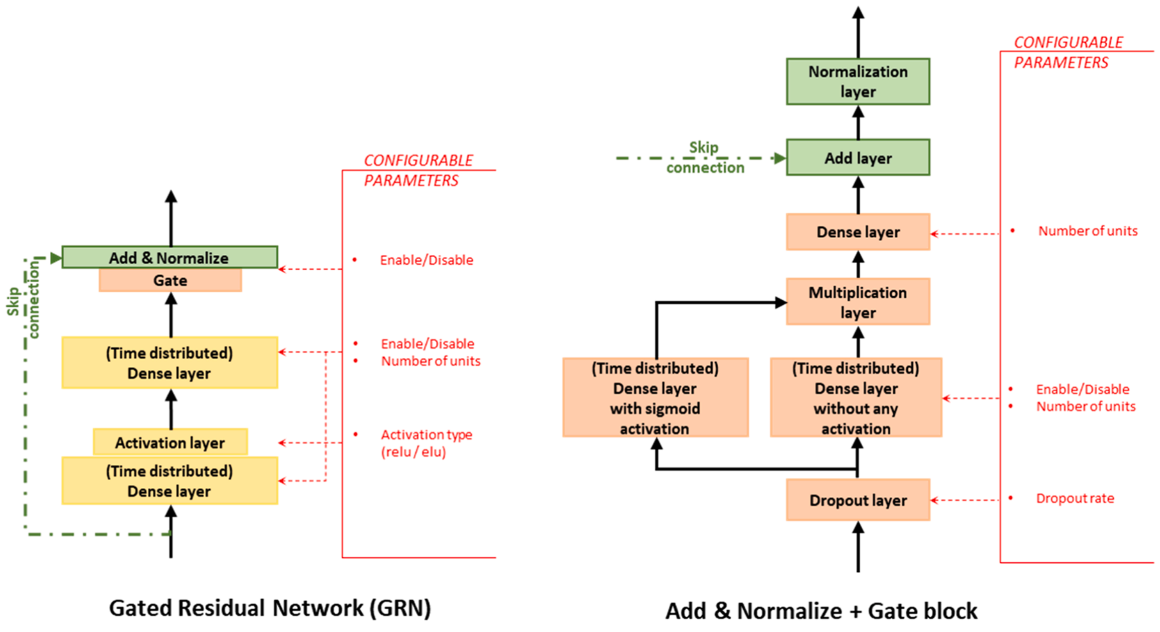


Fig. A.2. Schematic for the GRN and the gated *Add and Normalize* block components within the CONN model.

A.3 Temporal Convolutional Networks

Within the CONN architecture, TCNs are another pivotal tool for capturing the temporal dependencies of time series data. TCN, with its causal and dilated convolutions, and residual connections, effectively handles long sequences without the exploding gradient issues often encountered in recurrent networks [29,30,41]. Users have the flexibility to enable or disable the *TCN* block and modify its hyperparameters via the configuration file. Leveraging the Keras-TCN library [41] allows for efficient integration, providing another way for CONN to distil complex temporal patterns and enhance forecast quality.

Appendix B: Pseudocodes

Fig. B.1, Fig. B.2, Fig. B.3, Fig. B.4, Fig. B.5, Fig. B6

The following section provides simplified Python-like pseudocode representation of specific classes used in *pymodconn*. While they capture the essence of the main functionalities, note that the actual code is more nuanced, containing additional conditions and checks. This abstraction is intended to offer a clear, high-level perspective.

```
CLASS Model_Gen:
    INIT(cfg, current_dt):
        - IF 'if_seed' in cfg is True:
            - set seeds for numpy, TensorFlow and Keras for reproducibility
            - enable deterministic operations in TensorFlow
        - initialize various model parameters from cfg (like n_past, n_future, known_past_features, etc.)

    FUNCTION build_model():
        CREATE encoder input layer:
            - Shape is (n_past, known_past_features)
        PASS the encoder input through Encoder_class to get:
            - encoder_outputs_seq (sequence of hidden states)
            - encoder_outputs_allstates (final state of the encoder)
        CREATE decoder input layer:
            - Shape is (n_future, known_future_features)
        PASS decoder inputs, encoder_outputs_seq, and encoder_outputs_allstates through Decoder_class to get:
            - decoder_outputs (the main output of the decoder)
        IF model_type_prob is probabilistic ('prob'):
            - Adjust the shape of decoder_outputs using a Dense layer
            - Reshape the output to the target shape
            - IF using the parametric loss:
                - Apply soft ReLU activation for positive standard deviation
        ELSE IF model_type_prob is non-probabilistic ('nonprob'):
            - Clip the values of decoder_outputs between -1 and 1
        ELSE:
            - Raise an error for unsupported model_type_prob
        CONSTRUCT the Keras Model:
            - Inputs are [encoder_inputs, decoder_inputs]
            - Output is decoder_outputs4 (final reshaped and possibly activated output)
        Call Build_utils.postbuild_model method to:
            - Apply appropriate loss function, optimizer, and metrics based on the config
```

Fig. B.1. Pseudocode for *Model_gen*.

```
CLASS Encoder_class:
    INIT(cfg, enc_or_dec_number):
        - initialize various model parameters from cfg (like n_past, n_future, known_past_features, etc.)

    CALL FUNCTION(input, init_states=None):
        APPLY a Dense layer to the input:
            - This layer is used to adjust the number of features in the input to match the 'all_layers_neurons'
        APPLY Dropout to the Dense output:
            - The dropout rate is one fifth of 'all_layers_dropout'
        - Set 'output_cell' to the result from Dropout
        APPLY Temporal Convolutional Network (TCN) with addnorm and Gated Linear Units (GLU) at input:
            - It uses settings defined in the config under 'cfg['encoder']['TCN_' + self.location]'
        - Set 'input_cell' to 'output_cell' and update 'output_cell' with the result from the TCN layer
        APPLY Recurrent Neural Network (RNN) with addnorm and GLU at input:
            - It might use an optional 'init_states' if provided
        - Update 'output_cell' and 'output_states' with the results from the RNN layer
        - Set 'input_cell' to 'output_cell'
        APPLY Multi-Head Attention (MHA) with addnorm and GLU at input:
            - Loop for the depth of the MHA as specified in 'cfg['encoder']['self_MHA_block']['MHA_depth']':
                - Inside the loop, the MHA block is applied iteratively to 'input_cell'
                - Update 'output_cell' with the result and set 'input_cell' to 'output_cell'

        RETURN 'output' and 'output_states', where 'output' is the final processed tensor from the encoder
```

Fig. B.2. Pseudocode for *Encoder_class*.

```
CLASS Decoder_class:
    INIT(cfg, enc_or_dec_number):
        - Initialize various model parameters from cfg (like n_past, n_future, known_past_features, etc.)

    CALL FUNCTION(input, input_vk, encoder_states=None):
        PROCESS the input:
            - APPLY a Dense layer to adjust the number of features in the input to match 'all_layers_neurons'
            - APPLY Dropout layer using one fifth of 'all_layers_dropout'
            - SET 'output_cell' as the result
        APPLY Temporal Convolutional Network (TCN) with addnorm and Gated Linear Units (GLU) at input of decoder:
            - Set 'input_cell' to 'output_cell' and update 'output_cell' with the result from the TCN layer
        APPLY Recurrent Neural Network (RNN) with addnorm and GLU at input of decoder:
            - This step might use 'encoder_states' if provided
            - Update 'output_cell' and store the RNN states as 'decoder_input_states'
            - Set 'input_cell' to 'output_cell'
        APPLY Contextual Information Transfer (CIT_block) layer:
            - Update 'output_cell' by processing 'input_cell' and 'input_vk' through CIT_block
        APPLY Temporal Convolutional Network (TCN) with addnorm and GLU at output of decoder:
            - Set 'input_cell' to 'output_cell' and update 'output_cell' with the result from the TCN layer
        MERGE the encoder_input RNN states and decoder_input RNN states:
            - Use STATES_MANIPULATION_BLOCK to produce 'merged_states'
        APPLY Recurrent Neural Network (RNN) with addnorm and GLU at output of decoder:
            - Use 'merged_states' for initialization
            - Update 'output_cell' with the results from the RNN layer
        EXTRACT 'decoder_future' output:
            - APPLY TimeDistributed layer to 'output_cell' to produce the final decoder output

        RETURN the final decoder output
```

Fig. B.3. Pseudocode for *Decoder_class*.

```
CLASS CIT_block:
    INIT(cfg, enc_or_dec_number):
        - Initialize various parameters from cfg (like n_past, n_future, known_past_features, etc.)

    CALL FUNCTION(input_cell, input_enc):
        IF option is 1:
            - RESHAPE input_enc to match dimensions with n_future
            LOOP for the specified depth in cfg for option 1:
                - CONCATENATE input_cell with input_enc
                - APPLY a Dense layer to the concatenated tensor
                - BASED on the setting IF_NONE_GLUADDNORM_ADDNORM_CIT_1 from cfg:
                    IF it's set to 1:
                        - APPLY GLU_with_ADDNORM function
                    ELSE IF it's set to 2:
                        - APPLY ADD_NORM function
                - UPDATE input_cell with output_cell
        IF option is 2:
            LOOP for the specified depth in cfg for option 2:
                - DETERMINE attention type based on cfg
                - APPLY the appropriate Attention mechanism (either Attention or AdditiveAttention)
                - CONCATENATE the attention output with input_cell
                - APPLY a Dense layer to the concatenated tensor
                - BASED on the setting IF_NONE_GLUADDNORM_ADDNORM_CIT_2 from cfg:
                    IF it's set to 1:
                        - APPLY GLU_with_ADDNORM function
                    ELSE IF it's set to 2:
                        - APPLY ADD_NORM function
                - UPDATE input_cell with output_cell
        IF option is 3:
            IF cfg setting 'IF_SELF_CROSS_MHA' for decoder is 1:
                LOOP for the specified depth in cfg for SELF_CROSS_MHA:
                    - APPLY MHA_block_class for self attention
                    - UPDATE input_cell with output_cell from self attention
                    - APPLY MHA_block_class for cross attention
                    - UPDATE input_cell with output_cell from cross attention

        RETURN output_cell
```

Fig. B.4. Pseudocode for *CIT_block*.

```
CLASS MHA_block_class:
    INIT(cfg, enc_or_dec, enc_or_dec_number, self_or_crossMHA, mha_depth_index):
        - Extract various parameters from cfg (like IF_GRN_block, IF_MHA, mha_head, etc.)

    CALL FUNCTION(input_q, input_kv):
        IF Multi-head Attention is enabled:
            - Construct the name for the MHA layer
            - INITIALIZE a MultiHeadAttention layer with the required configurations
            - APPLY this MHA layer to the input query, key, and value
            BASED on the setting IF_NONE_GLUADDNORM_ADDNORM from cfg:
                IF it's set to 1:
                    - APPLY GLU_with_ADDNORM function to the result
                ELSE IF it's set to 2:
                    - APPLY ADD_NORM function to the result
            IF the GRN block condition is satisfied:
                - APPLY the GRN_layer with required configurations to the result
        ELSE:
            - Set output_cell to be the same as input_q

        RETURN output_cell
```

Fig. B.5. Pseudocode for *MHA_block_class*.

```
CLASS RNN_block_class:
    INIT(cfg, enc_or_dec, input_or_output, num):
        - ensure input_or_output is either 'input' or 'output'
        - initialize various configuration parameters (like IF_RNN, IF_NONE_GLUADDNORM_ADDNORM, IF_GRN)

    CALL FUNCTION(input_cell, init_states=None):
        IF RNN is enabled in configuration:
            - Create a rnn_unit object
            - Use this object to process the input_cell and potentially use init_states
            - Extract the output tensor and state tensors
            - Apply transformations on the output tensor based on configurations:
                - Apply a linear layer
                - Apply Gated Linear Units (GLU) with Add-Normalization
                - Apply a linear layer followed by Add-Normalization
            - If Gated Residual Network (GRN) is enabled in configuration, apply GRN layer to the output tensor
        ELSE:
            - Directly assign input_cell to output_cell and set output_states to None

        RETURN output_cell and output_states

CLASS rnn_unit:
    INIT(cfg, enc_or_dec, input_or_output, num):
        - initialize various configuration parameters

    CALL FUNCTION(input_cell, init_states=None):
        IF the rnn_depth is set to 1:
            - Return the result of processing input_cell using single_rnn_layer
        ELSE:
            - Process the input_cell using single_rnn_layer and save the result to 'x'
            - Apply transformations on 'x' based on configurations:
                - Apply Gated Linear Units (GLU) with Add-Normalization
                - Apply a linear layer followed by Add-Normalization
            - For remaining RNN layers (depth - 2), repeat the above process
            - Finally, process 'x' using single_rnn_layer and return the result

    FUNCTION single_rnn_layer(x_input, init_states, mid_layer, layername_prefix):
        - Based on the RNN type configuration, determine the RNN layer to be used (LSTM, GRU, or SimpleRNN)
        - Setup initial states if required
        - Based on the position of the layer in the sequence (beginning, middle, end), set flags for sequence and state returns
        - If the RNN is bidirectional, wrap the RNN layer in a Bidirectional wrapper
        - Process the x_input using the RNN layer
        - Return the processed output
```

Fig. B.6. Pseudocode for *RNN_block_class* and *rnn_unit*.

Appendix C: Implementation example

After installation as described in Section 3, the package can be imported into a Python program. An implementation example is presented in Fig. C.1. This script tests the three different example model types shown in Section 3. The details of the script are elaborated in this Appendix section.

Synthetic time series data is generated using *numpy* for training and evaluation. Specifically, random input sequences with five known past features, three known future features, and two unknown future features are created with 1000 samples, a past observation window of 25 steps, and a future prediction window of 10 steps. The data is split into training and test sets using a specified percentage split.

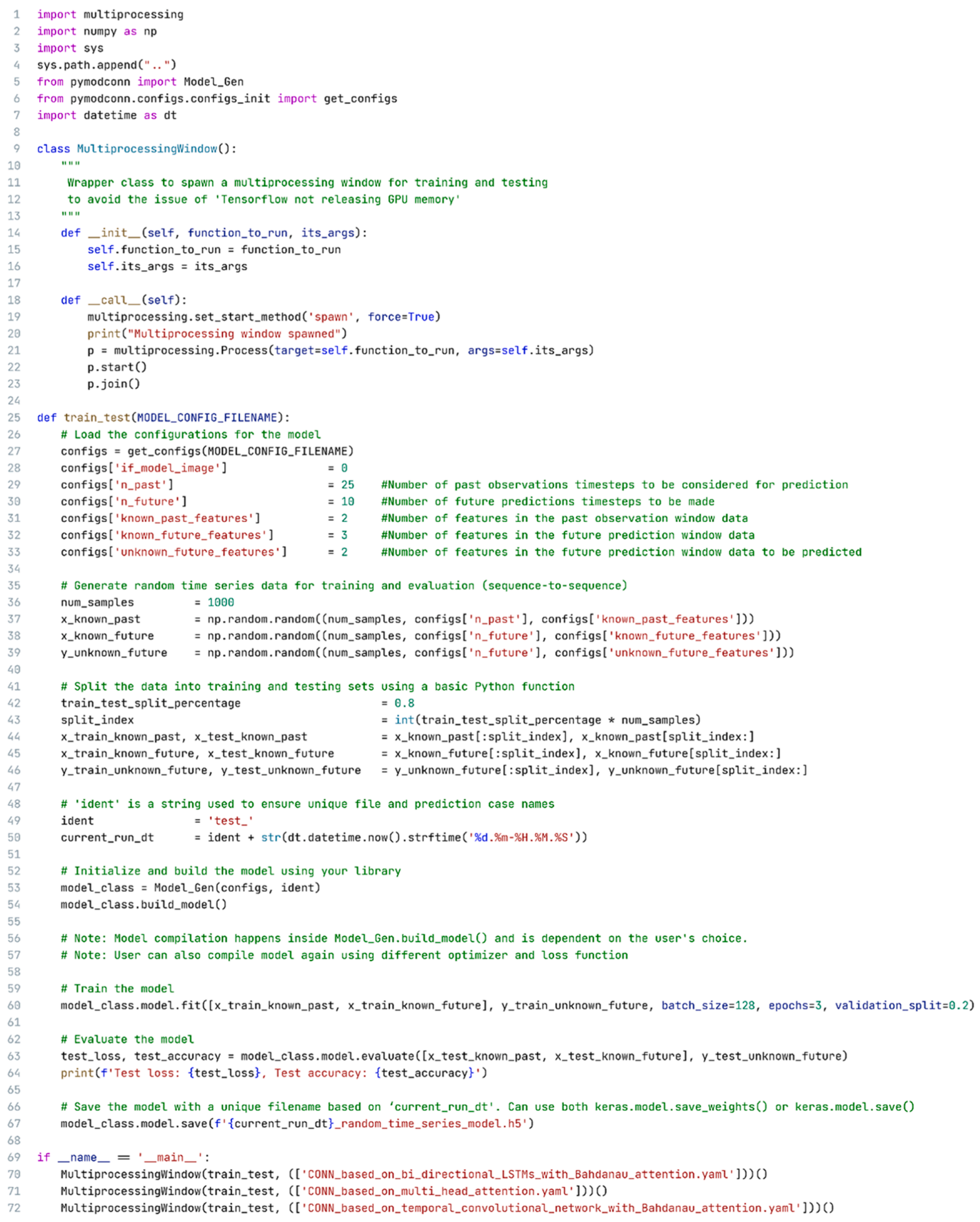
The *get_configs()* function from the *pymodconn.configs_init* module is then used to load the configurations for the model from a *.yaml* file as a dictionary. This facilitates implementation of various network architectures, promotes modularity and enables the reuse of predefined models, thus improving efficiency and reducing coding errors. The approach simplifies hyperparameter tuning, a critical aspect of neural network performance, by isolating these parameters in a configuration file. This file can be generated by providing *config_filename = None* in the *get_configs(config_filename)* function or obtained from the GitHub repository of the package (https://github.com/gaurav306/pymodconn/blob/master/pymodconn/configs/default_config.yaml). With the former approach, a template of the configuration file is created in the directory of the Python program, which can be modified by the users. With the latter approach, users can download a template configuration file, along with an assortment of configuration files corresponding to various predefined model architectures. The users can use configuration files for predefined model architectures as starting points. To avoid errors and ensure correct operation, it is recommended that users do not alter the key names or the data types within the configuration file. Maintaining this structure is essential for *pymodconn* to function correctly. Fig. C.2 shows a part of the configuration file meant to configure components of the *Encoder* block.

Going back to implementation example, a unique identifier string (*ident*) is created by appending the current time string to a predefined text. This identifier is used to differentiate between multiple runs or cases and for generating unique filenames when saving the model.

The *Model_Gen* class is instantiated with the configuration file dictionary and the unique run identifier. The *build_model()* function is then called to create the model object. It is important to note that the *model.compile()* function in the *build_model()* method is dependent on the users' choice of values for the parameters. For point-based forecasts, the users can decide not to include *model.compile()* inside the *model_class.build_model()* method. This gives the users more control over the available compile options, such as loss functions, error metrics, and learning rate schedulers. In contrast, for probabilistic forecasts, the *model.compile()* function is included inside the *model_class.build_model()* method, as custom loss functions are used for probabilistic forecasting.

The model returned from *build_model()* can be used similarly to a Keras-based model. The users can run *model.fit(), model.train_on_batch(), model.*

evaluate(), or *model.save()* as needed. The users can also check the summary of the model using *model.summary()* and save the plot image using *keras.utils.vis_utils.plot_model()*.



```python
import multiprocessing
import numpy as np
import sys
sys.path.append("..")
from pymodconn import Model_Gen
from pymodconn.configs.configs_init import get_configs
import datetime as dt

class MultiprocessingWindow():
    """
     Wrapper class to spawn a multiprocessing window for training and testing
     to avoid the issue of 'Tensorflow not releasing GPU memory'
    """
    def __init__(self, function_to_run, its_args):
        self.function_to_run = function_to_run
        self.its_args = its_args

    def __call__(self):
        multiprocessing.set_start_method('spawn', force=True)
        print("Multiprocessing window spawned")
        p = multiprocessing.Process(target=self.function_to_run, args=self.its_args)
        p.start()
        p.join()

def train_test(MODEL_CONFIG_FILENAME):
    # Load the configurations for the model
    configs = get_configs(MODEL_CONFIG_FILENAME)
    configs['if_model_image']             = 0
    configs['n_past']                     = 25    #Number of past observations timesteps to be considered for prediction
    configs['n_future']                   = 10    #Number of future predictions timesteps to be made
    configs['known_past_features']        = 2     #Number of features in the past observation window data
    configs['known_future_features']      = 3     #Number of features in the future prediction window data
    configs['unknown_future_features']    = 2     #Number of features in the future prediction window data to be predicted

    # Generate random time series data for training and evaluation (sequence-to-sequence)
    num_samples         = 1000
    x_known_past        = np.random.random((num_samples, configs['n_past'], configs['known_past_features']))
    x_known_future      = np.random.random((num_samples, configs['n_future'], configs['known_future_features']))
    y_unknown_future    = np.random.random((num_samples, configs['n_future'], configs['unknown_future_features']))

    # Split the data into training and testing sets using a basic Python function
    train_test_split_percentage                     = 0.8
    split_index                                     = int(train_test_split_percentage * num_samples)
    x_train_known_past, x_test_known_past           = x_known_past[:split_index], x_known_past[split_index:]
    x_train_known_future, x_test_known_future       = x_known_future[:split_index], x_known_future[split_index:]
    y_train_unknown_future, y_test_unknown_future   = y_unknown_future[:split_index], y_unknown_future[split_index:]

    # 'ident' is a string used to ensure unique file and prediction case names
    ident               = 'test_'
    current_run_dt      = ident + str(dt.datetime.now().strftime('%d.%m-%H.%M.%S'))

    # Initialize and build the model using your library
    model_class = Model_Gen(configs, ident)
    model_class.build_model()

    # Note: Model compilation happens inside Model_Gen.build_model() and is dependent on the user's choice.
    # Note: User can also compile model again using different optimizer and loss function

    # Train the model
    model_class.model.fit([x_train_known_past, x_train_known_future], y_train_unknown_future, batch_size=128, epochs=3, validation_split=0.2)

    # Evaluate the model
    test_loss, test_accuracy = model_class.model.evaluate([x_test_known_past, x_test_known_future], y_test_unknown_future)
    print(f'Test loss: {test_loss}, Test accuracy: {test_accuracy}')

    # Save the model with a unique filename based on 'current_run_dt'. Can use both keras.model.save_weights() or keras.model.save()
    model_class.model.save(f'{current_run_dt}_random_time_series_model.h5')

if __name__ == '__main__':
    MultiprocessingWindow(train_test, (['CONN_based_on_bi_directional_LSTMs_with_Bahdanau_attention.yaml']))()
    MultiprocessingWindow(train_test, (['CONN_based_on_multi_head_attention.yaml']))()
    MultiprocessingWindow(train_test, (['CONN_based_on_temporal_convolutional_network_with_Bahdanau_attention.yaml']))()
```

Fig. C1. Example implementation of *pymodconn*.

```
# DETAILED MODEL SETTINGS
# ENCODER SETTINGS
encoder:
  TCN_input:
    IF_TCN: 0
    IF_NONE_GLUADDNORM_ADDNORM_TCN: 1           # 0: None, 1: GLUADDNORM, 2: ADDNORM
    kernel_size: 3
    nb_stacks: 2
    dilations: [1, 2, 4, 8, 16, 32]

  RNN_block_input:
    # settings for RNN block outside the block class
    IF_RNN: 1
    IF_NONE_GLUADDNORM_ADDNORM_block: 1         # 0: None, 1: GLUADDNORM, 2: ADDNORM
    IF_GRN_block: 1                             # 0: no GRN, 1: GRN
    # settings for RNN units inside the blocks
    rnn_depth: 8
    rnn_type: LSTM                              # GRU, LSTM, SimpleRNN
    IF_birectionalRNN: 1                        # 0: Unidirectional, 1: Bidirectional
    IF_NONE_GLUADDNORM_ADDNORM_deep: 1          # 0: None, 1: GLUADDNORM, 2: ADDNORM

  self_MHA_block:
    # settings for self MHA block outside the block class
    MHA_depth: 3
    # settings for self MHA units inside the blocks
    IF_MHA: 0
    IF_GRN_block: 1                             # 0: no GRN, 1: GRN
    MHA_head: 8
    IF_NONE_GLUADDNORM_ADDNORM_deep: 1          # 0: None, 1: GLUADDNORM, 2: ADDNORM
```

Fig. C2. Example snippet of a configuration file showing how the *Encoder* block can be configured.

References

[1] Mata É, Peñaloza D, Sandkvist F, Nyberg T. What is stopping low-carbon buildings? A global review of enablers and barriers. Energy Research and Social Science 2021;82.
[2] Drgoňa J, Arroyo J, Cupeiro Figueroa I, Blum D, Arendt K, Kim D, Ollé EP, Oravec J, Wetter M, Vrabie DL, Helsen L. All you need to know about model predictive control for buildings. Annual Reviews in Control 2020;50:190–232.
[3] Chen Y, Tong Z, Zheng Y, Samuelson H, Norford L. Transfer learning with deep neural networks for model predictive control of HVAC and natural ventilation in smart buildings. J Cleaner Prod 2020;254:119866.
[4] Troitzsch S. and Hamacher T. 2020 Control-oriented Thermal Building Modelling.
[5] Boersma S, Doekemeijer B, Vali M, Meyers J, van Wingerden JW. A control-oriented dynamic wind farm model. WFSim Wind Energy Science 2018;3:75–95.
[6] Riemann BJC, Li J, Adewuyi K, Landers RG, Park J. Control-Oriented Modeling of Lithium-Ion Batteries Journal of Dynamic Systems. Measurement, and Control 2020:143.
[7] Yao J, Han T. Data-driven lithium-ion batteries capacity estimation based on deep transfer learning using partial segment of charging/discharging data. Energy 2023;271:127033.
[8] Yu X, Georges L, Imsland L. Data pre-processing and optimization techniques for stochastic and deterministic low-order grey-box models of residential buildings. Energy Build 2021;236:110775.
[9] Arendt K, Jradi M, Shaker HR, Veje CT. Comparative analysis of white-, gray- And black-box models for thermal simulation of indoor environment. In: Teaching building case study ASHRAE and IBPSA-USA Building Simulation Conference; 2018. p. 173–80. pp.
[10] de Souza AK, Hileman W, Trimboli MS, Plett GL. A Control-Oriented Reduced-Order Model for Lithium-Metal Batteries. IEEE Control Systems Letters 2022;7:1165–70.
[11] Johra H, Schaffer M, Chaudhary G, Kazmi HS, Le Dréau J, Petersen S. What Metrics Does the Building Energy Performance Community Use to Compare Dynamic Models?. In: Proceedings of Building Simulation 2023: 18th Conference of International Building Performance Simulation Association. IBPSA; 2023. 4-6 September 2023.
[12] Guo J, He H, Sun C. ARIMA-based road gradient and vehicle velocity prediction for hybrid electric vehicle energy management. IEEE Trans Veh Technol 2019;68:5309–20.
[13] Aliramezani M, Norouzi A, Koch CR. Support vector machine for a diesel engine performance and NOx emission control-oriented model. IFAC-PapersOnLine 2020;53:13976–81.
[14] Gokhale G, Claessens B, Develder C. Physics informed neural networks for control oriented thermal modeling of buildings. Appl Energy 2022;314:118852.
[15] Li S, Li W, Noack BR. Machine-learned control-oriented flow estimation for multi-actuator multi-sensor systems exemplified for the fluidic pinball. J Fluid Mech 2022;952:A36.
[16] Qi D, Majda AJ. Using machine learning to predict extreme events in complex systems. Proc Natl Acad Sci 2020;117:52–9.
[17] Rajendra P, Brahmajirao V. Modeling of dynamical systems through deep learning. Biophys Rev 2020;12:1311–20.
[18] Torres JM, Aguilar RM. Using Deep Learning to Predict Complex Systems: A Case Study in Wind Farm Generation ed J M Andújar. complex 2018:9327536. 2018.
[19] Hornik K. Approximation capabilities of multilayer feedforward networks. Neural Netw 1991;4:251–7.
[20] Ruano AE, Crispim EM, Conceiçao EZ, Lúcio MMJ. Prediction of building's temperature using neural networks models. Energy Build 2006;38:682–94.
[21] Jin H, Chollet F, Song Q, Hu X. AutoKeras: An AutoML Library for Deep Learning. J Mach Learn Res 2023;24:1–6.
[22] Wood L., Tan Z., Stenbit I., Bischof J., Zhu S., Chollet F., and others 2022 KerasCV.
[23] Shen W. DeepCTR: easy-to-use,modular and extendible package of deep-learning based ctr models. GitHub Repository; 2017.
[24] Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T. Caffe: convolutional architecture for fast feature embedding. arXiv preprint; 2014.
[25] Howard J, Gugger S. Fastai: A Layered API for Deep Learning. Information 2020;11.
[26] Ramsundar B., Eastman P., Walters P., Pande V., Leswing K. and Wu Z. 2019 Deep Learning for the Life Sciences (O'Reilly Media).

[27] Abadi M., Agarwal A., Barham P., Brevdo E., Chen Z., Citro C., Corrado G. S., Davis A., Dean J. and Devin M. 2016 Tensorflow: large-scale machine learning on heterogeneous distributed systems arXiv preprint.

[28] Sutskever I, Vinyals O, Le QV. Sequence to sequence learning with neural networks. In: Advances in neural information processing systems; 2014. p. 27.

[29] Chen Y, Kang Y, Chen Y, Wang Z. Probabilistic forecasting with temporal convolutional neural network. Neurocomputing 2020;399:491–501. %@ 0925-2312.

[30] Bai S, Kolter JZ, Koltun V. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. arXiv preprint; 2018.

[31] Rumelhart DE, Hinton GE, Williams RJ. Learning internal representations by error propagation. California Univ San Diego La Jolla Inst for Cognitive Science; 1985.

[32] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput 1997;9: 1735–80.

[33] Chung J, Gulcehre C, Cho K, Bengio Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint; 2014.

[34] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. arXiv preprint; 2014.

[35] Luong MT, Pham H, Manning CD. Effective approaches to attention-based neural machine translation. arXiv preprint; 2015.

[36] Lim B, Arık SÖ, Loeff N, Pfister T. Temporal fusion transformers for interpretable multi-horizon time series forecasting. Int J Forecast 2021;37:1748–64. %@ 0169-2070.

[37] Chaudhary G. 2023 gaurav306/pymodconn: pymodconn : A Python package for developing modular sequence to sequence control oriented neural networks.

[38] Chaudhary G, Johra H, Georges L, Austbø B. Predicting the performance of hybrid ventilation in buildings using a multivariate attention-based biLSTM encoder-decoder neural network. arXiv preprint; 2023.

[39] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Advances in neural information processing systems; 2017. p. 30.

[40] He K., Zhang X., Ren S. and Sun J. 2016 Deep residual learning for image recognition pp 770–8.

[41] Remy P. Temporal Convolutional Networks for Keras. GitHub repository 2020.