Contents lists available at ScienceDirect

SoftwareX

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

Original software publication

GNNkeras: A Keras-based library for Graph Neural Networks and homogeneous and heterogeneous graph processing

Niccolò Pancino [a,b,*], Pietro Bongini [a,b], Franco Scarselli [a], Monica Bianchini [a]

[a] *University of Siena, Department of Information Engineering and Mathematics, Via Roma 56, 53100, Siena (SI), Italy*
[b] *University of Florence, Department of Information Engineering, Via S. Marta 3, 50139, Florence (FI), Italy*

ARTICLE INFO

Article history:
Received 18 January 2022
Received in revised form 27 February 2022
Accepted 17 March 2022

Keywords:
Graphs
Machine Learning
TensorFlow
Keras
Graph Neural Networks

ABSTRACT

In several areas of science and engineering, data can be naturally represented in graph form, where nodes denote entities and edges stand for relationships between them. Graph Neural Networks (GNNs) are a well-known class of machine learning models for graph processing. In this paper, we present GNNkeras, a library, based on Keras, which allows the implementation of a large subclass of GNNs. GNNkeras is a flexible tool: the implemented models can be used to classify/cluster nodes, edges, or whole graphs. Moreover, GNNkeras can be applied to both homogeneous and heterogeneous graphs, exploiting both inductive and mixed inductive–transductive learning, and can implement a layered version of GNNs, namely the LGNN model.

Code metadata

Current code version	v2.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00019
Code Ocean compute capsule	
Legal Code License	BSD 3-Clause Licence
Code versioning system used	git
Software code languages, tools, and services used	python3
Compilation requirements, operating environments & dependencies	TensorFlow 2.x, NumPy, SciPy, Typing
If available Link to developer documentation/manual	github.com/NickDrake117/GNNkeras/tree/GNNkeras_SoftwareX
Support email for questions	niccolo.pancino@unifi.it

Software metadata

Current software version	2.0
Permanent link to executables of this version	github.com/NickDrake117/GNNkeras/tree/GNNkeras_SoftwareX
Legal Software License	BSD 3-Clause License
Computing platforms/Operating Systems	Linux, OS X, Microsoft Windows
Installation requirements & dependencies	TensorFlow 2.x, NumPy, SciPy, Typing
If available, link to user manual - if formally published include a reference to the publication in the reference list	github.com/NickDrake117/GNNkeras/tree/GNNkeras_SoftwareX
Support email for questions	niccolo.pancino@unifi.it

1. Motivation and significance

A graph is a data structure composed of a collection of nodes and edges that can be used to represent objects or patterns along with their relationships. Nodes and edges can be associated with vectors of values, describing the attributes of patterns

[*] Corresponding author at: University of Siena, Department of Information Engineering and Mathematics, Via Roma 56, 53100, Siena (SI), Italy.
E-mail address: niccolo.pancino@unifi.it (Niccolò Pancino).

https://doi.org/10.1016/j.softx.2022.101061

and relationships, respectively. Nowadays, graphs play an important role in many modern applications, since they are widely used to describe the information of interest in many real-world problems in different fields, including physics, social science, economics, and bioinformatics. For instance, in biological and chemical processes, nodes denote entities, such as atoms, proteins, or genes, while edges represent chemical bonds, physical contacts, or metabolic interactions. Actually, graphs constitute the natural data domain in many bioinformatics applications, such as, for instance, in the identification of interfacing amino acids in Protein–Protein Interaction (PPI) [1–3] or in the prediction of polypharmacy side effects [4]. More generally, it can be observed that a graph representation allows to naturally merge the information from different applications, by joining the corresponding graphs. In fact, in the extreme case, the entire information relating to the set of applications owned by a company/organization can be collected in a single graph.

Common Machine Learning (ML) techniques can only be applied to vector data, but the advancement of the role played by relationships in modern applications prompted researchers to design new approaches for the graph-structured data domain. Graph Neural Networks (GNNs) are a well-known class of machine learning models for graph-structured data processing based on neural networks. The first GNN model has been proposed in [5]. Recently, a huge number of approaches has been introduced, including Graph Convolutional Networks [6], GrapSAGE [7], Graph Attention Networks [8], and Graph Networks [9]. Moreover, GNNs have been applied to a large number of tasks, such as for drug repurposing [10], physics simulation [11], and recommendation systems [12]. See [13–15] for some recent reviews.

In this context, it is important for researchers and software developers to have adequate and flexible tools that support the development of applications with current GNN models and possibly favor the study of new versions of GNNs. While there are many deep learning frameworks available today, Keras along with TensorFlow 2 has greater adoption in both the industry and the research community than any other deep learning solution. Keras provides optimized modules, it is scalable and it has native support for mixed-precision training on Nvidia GPUs and TPUs for speeding up the learning process. For this reason, we developed a new Keras library which consents to implement an important subclass of GNNs. More precisely, GNNkeras provides a GNN model derived from the original one proposed in [5]: this model can be easily extended to all the recurrent GNNs [15], which are a large subclass of GNNs. The Layered Graph Neural Networks (LGNNs) are also available [16]: LGNNs can be considered a deeper version of GNNs and are capable of overcoming the so-called long-term dependency problem, i.e. the inability of the network to correctly process complex graphs, due to dependencies between distant nodes.

GNNkeras users can of course easily access a huge number of ML features. This fact is guaranteed by Keras itself, which is built on top of TensorFlow 2, one of the most used and complete software libraries for ML. Also note that, as far as we know, GNNkeras is the first tool specifically designed for recurrent GNNs.[1]

Finally, GNNkeras is flexible and permits to manage a variety of activities, graph domains, and learning approaches. In fact, GNNkeras can tackle tasks where the goal is to classify patterns, the relationships represented by the edges in a graph, or even an object represented by an entire graph. Moreover, both homogeneous and heterogeneous graph-structured data can be processed. Heterogeneous graphs, where nodes/edges represent different types of objects and can have different features, are especially important in modern applications, where information is often gathered from different sources. Ultimately, GNNkeras allows the use of two types of learning paradigms: a classical inductive learning scheme and a mixed inductive–transitive learning scheme [17,18], in which, for some nodes, features are enriched with the corresponding target,[2] which is explicitly exploited in the diffusion process and provides a direct transductive contribution.

2. Software description

The GNNkeras software is based on TensorFlow 2.x and Keras (TensorFlow backend), one of the most used deep learning frameworks worldwide [19]. Three types of tasks can be faced, called node-focused, edge-focused, and graph-focused. Node-focused problems concern situations in which all the nodes of a graph, or a subset of them, have a desired target: intuitively, an output must be produced in correspondence of each targeted node, which can be used for classification, regression, or clustering purposes. For example, localize a particular compound in a macromolecule, when the molecule is represented as a graph, is a node-focused task. On the other hand, edge-focused problems concern tasks in which the targets are associated to the edges: the GNN must classify, cluster, or even predict the existence of relationships between patterns. Predicting the nature of chemical bonds between atoms or molecules represents an edge-focused task. Finally, a graph-focused task concerns problems in which a unique target is associated to the whole graph, and the goal is to predict a property or to cluster the complex object represented by the graph. Predicting the mutagenicity of a particular compound is an example of this kind of task.

Based on an information diffusion mechanism, GNNs can process homogeneous and heterogeneous graph-structured data. In particular, GNNs create an *encoding network*, an architecture which replicates the topology of the input graph, by means of Multi-Layer Perceptron (MLP) units, implementing a state transition function f_w at each node, and an output function g_w (on targeted nodes or edges). The network unfolds itself in time and space, respectively, by replicating the MLP units on each node of the input graph, and by recurrently exchanging neighborhood information until a stable equilibrium point or the maximum number of iterations is reached. In this resulting feed-forward network, called *unfolding network*, each level corresponds to an instant in time and contains a copy of all the elements of the encoding network, on which the connections between the various layers also depend.

In the heterogeneous graph-structured data domain, the learning process differs from the one in the homogeneous case only for the number of MLPs used as building blocks, since different MLPs are exploited on the states of different types of nodes. In the heterogeneous setting, indeed, there is a different state updating MLP for each node or edge type, which will learn different versions of the state transition function. An example of the learning process on a heterogeneous graph is depicted in Fig. 1.

In the original framework, GNNs are inductively trained based on a supervised learning environment. However, GNNs and LGNNs can also take advantage of transductive learning [17,18], thanks

[1] More precisely, at the time of submission of this manuscript, an alpha version of an official GNN library for Keras is available at https://github.com/tensorflow/gnn, which, however, appears to be designed primarily for convolutional rather than for recurrent GNNs, like the ones offered by our software.

[2] In this paper, we identify with the term *target* the expected output of the network on a certain entity, a node for example. Therefore, in the case of a node-focused classification problem, the target will represent the class to which the node belongs.

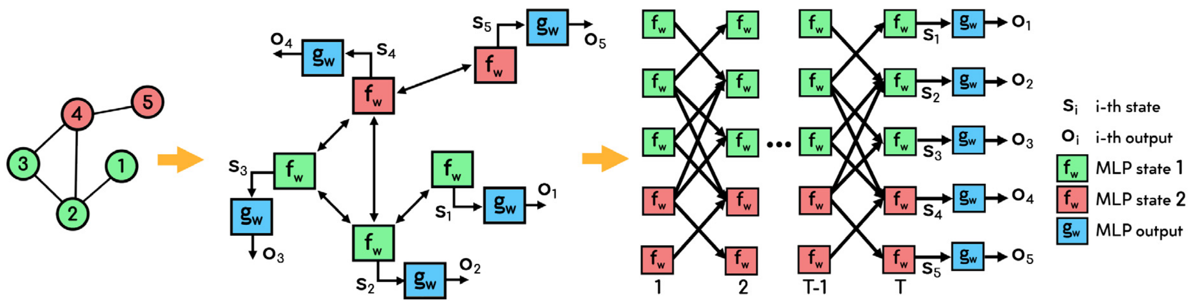


Fig. 1. A composite GNN learning on a heterogeneous graph with two node types (green and red) and one edge type. From left to right: a generic undirected input graph; the encoding network, where green and red blocks represent two types of state transition function f_w, while blue boxes represent the output function g_w; the unfolding network, in the form of a feed-forward network with $T+1$ layers. The same scheme applies to a non-composite GNN with a unique MLP implementing f_w on each node. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

to the natural way the information flows and spreads across the graph. In the transductive framework, the training set nodes and their targets are used in conjunction with the test patterns. In particular, the feature vectors of a subset of the training nodes – called transductive nodes – are enriched with their targets, to be explicitly exploited in the diffusion process, yielding a direct transductive contribution.

2.1. Definitions

Formally, a generic graph is a pair $G = (V, E)$, where $V = \{v_0, v_1, \dots, v_{n-1}\}$ denotes the set of *vertices* or *nodes*, and $E = \{(v_i, v_j) : v_i, v_j \in V\}$ represents the set of its *arcs*. In particular, $v_i \in V$ denotes a node and $e_{ij} \equiv (v_i, v_j) \in E$ denotes an arc connecting v_i to v_j and pointing from v_i to v_j. More specifically, when the graph is undirected, the connection between two nodes is called an *edge* and, in our implementation, it corresponds to two arcs (v_i, v_j) and (v_j, v_i), pointing in opposite directions. The order and the size of a graph are defined as its number of nodes $n = |V|$ and its number of arcs $m = |E|$, respectively. The set $\text{in}_v(v_i) = \{v_j \in V : e_{ji} \in E\}$ is the incoming neighborhood of v_i, i.e. its adjacent nodes, connected to v_i by an arc pointing to v_i, while $\text{in}_e(v_i) = \{e_{ji} \in E\}$ is the set of arcs entering v_i, that is, coming from its neighborhood and pointing to v_i. The size of the incoming neighborhood $|\text{in}_v(v_i)|$ for a given node v_i corresponds to the number of its neighbors in in_v. Moreover, a generic finite graph can be represented by its *adjacency matrix* $\mathbf{A} \in \mathbb{R}^{n\times n} = \{a_{ij} \in \{0, 1\} : 0 \le i \le n-1, 0 \le j \le n-1\}$ whose elements indicate whether or not two nodes are adjacent in the graph. GNNkeras exploits a modified version of this matrix, based on some criteria: in particular, if there exists an arc $(v_i, v_j) \in E$, connecting two generic nodes $v_i, v_j \in V$, then $a_{ij} \in \mathbb{R} \neq 0$, otherwise $a_{ij} = 0$.

Nodes and arcs can be respectively associated with vectors $\mathbf{x}_i \in \mathbb{R}^{l_n}$ and $\mathbf{e}_{ij} \in \mathbb{R}^{l_e}$ of attributes describing their features. By stacking all these vectors, the *node feature matrix* $\mathbf{X} \in \mathbb{R}^{n\times l_n}$, and the *arc feature matrix* $\mathbf{E} \in \mathbb{R}^{m\times(2+l_e)}$ are obtained: a generic row $\mathbf{x}_i \in \mathbf{X}$ represents the i-th node feature vector, while a generic row $\mathbf{e}_i \in \mathbf{E}$ describes the i-th arc, linking the nodes whose indices in $\mathbf{X}$ constitute the first two components of $\mathbf{e}_i$, i.e. $\mathbf{e}_i^0$ and $\mathbf{e}_i^1$, thus pointing from the node $v_{\mathbf{e}_i^0}$ to $v_{\mathbf{e}_i^1}$.

A representation of the node $v_i \in V$ can be provided by its *state* $\mathbf{s}_i \in \mathbb{R}^{d_s}$, obtained based on the information contained in v_i as well as in its incoming neighborhood $\text{in}_v(v_i)$. By stacking all the states, a node-state matrix $\mathbf{S} \in \mathbb{R}^{n\times d_s}$ is collected.

During graph processing, the state of all the nodes is updated iteratively, until convergence or a maximum number of iterations is reached at step T. During this process, each node interacts with its neighbors by exchanging messages. Let f_w be a parametric *state transition* function, used for the state calculation, g_w be a parametric *output* function, which describes how the output is produced at step $t = T$, and φ be a *neighborhood aggregation* function, which defines how the messages coming from $\text{in}_v(v_i)$ – composed of nodes' feature vectors, nodes' states and arcs' states – are aggregated. Then the state $\mathbf{s}_i^t$ for $t \le T$ and the output $\mathbf{o}_i^T = \mathbf{o}_i$ at node v_i are defined as follows:

$$\begin{aligned} \mathbf{s}_i^t &= f_w(\mathbf{s}_i^{t-1},\ \mathbf{x}_i,\ \varphi(\mathbf{s}^{t-1}_{\text{in}_v(v_i)}, \mathbf{x}_{\text{in}_v(v_i)}, \mathbf{e}_{\text{in}_e(v_i)})) \\ \mathbf{o}_i &= g_w(\mathbf{s}_i^T, \mathbf{x}_i) \end{aligned} \tag{1}$$

Note that the state $\mathbf{s}_i^t$ for the node v_i is based on its features, on its state at the previous time step $\mathbf{s}_i^{t-1}$ and on an aggregated message derived from its incoming neighborhood. Eq. (1) can be rewritten in a compact form as:

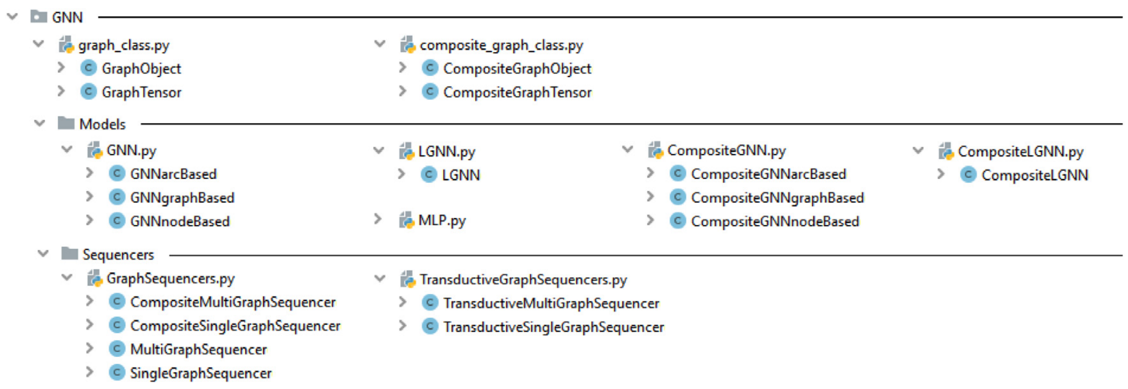
$$\begin{aligned} \mathbf{S}^t &= F_w(\mathbf{S}^{t-1},\ \mathbf{X},\ \varphi(\mathbf{S}^{t-1}, \mathbf{X}, \mathbf{E})) \\ \mathbf{O} &= G_w(\mathbf{S}^T, \mathbf{X}) \end{aligned} \tag{2}$$

where F_w and G_w are the *global* transition function and *global* output function, i.e. the stacked version of $|V|$ instances of f_w and g_w, respectively. Both f_w and g_w can be implemented by MLP units, in the following referred to as *net*$_s$ and *net*$_o$. Function φ can be any aggregation function of the messages from $\text{in}_v(v_i)$. In particular, three possible functions φ are considered in the software: the sum of the incoming messages, φ_{sum}, the average over the size of the incoming neighborhood $\text{in}_v(v_i)$, φ_{avg}, and the normalization over the size of the graph, φ_{norm}, defined as:

$$\begin{aligned} \varphi_{sum}(v_i) &= \sum_{j\in \text{in}_v(v_i)} (\mathbf{s}_j^{t-1}, \mathbf{x}_j, \mathbf{e}_{ji}) \\ \varphi_{avg}(v_i) &= \frac{1}{|ne(v_i)|} \sum_{j\in \text{in}_v(v_i)} (\mathbf{s}_j^{t-1}, \mathbf{x}_j, \mathbf{e}_{ji}) \\ \varphi_{norm}(v_i) &= \frac{1}{|V|} \sum_{j\in \text{in}_v(v_i)} (\mathbf{s}_j^{t-1}, \mathbf{x}_j, \mathbf{e}_{ji}) \end{aligned} \tag{3}$$

In supervised and semi-supervised applications, a generic graph is also associated with a *target matrix* $\mathbf{T}$, which is obtained by stacking all the available target values and whose dimensions are variable, since it can be associated with the whole graph, the whole set of nodes or arcs, or a subset of them.

In the heterogeneous graph domain, both V and E could represent objects and relations of different type or nature, sometimes also being described by feature vectors of variable length. Let K be a finite set of node types, associated with a node feature set dimension $L_n^K = \{l_n^k : k \in K\}$, then $v_i^k \in V$ denotes a generic node of the graph belonging to type $k \in K$ and described by a feature vector $\mathbf{x}_i^k \in \mathbb{R}^{l_n^k}$. Although for the sake of completeness, arcs representing different types of relations can be described by feature vectors of different lengths, in the vast majority of heterogeneous graph datasets, they are not related to any attribute: in



```
GNN
  graph_class.py
    GraphObject
    GraphTensor
  composite_graph_class.py
    CompositeGraphObject
    CompositeGraphTensor
  Models
    GNN.py
      GNNarcBased
      GNNgraphBased
      GNNnodeBased
    LGNN.py
      LGNN
    MLP.py
    CompositeGNN.py
      CompositeGNNarcBased
      CompositeGNNgraphBased
      CompositeGNNnodeBased
    CompositeLGNN.py
      CompositeLGNN
  Sequencers
    GraphSequencers.py
      CompositeMultiGraphSequencer
      CompositeSingleGraphSequencer
      MultiGraphSequencer
      SingleGraphSequencer
    TransductiveGraphSequencers.py
      TransductiveMultiGraphSequencer
      TransductiveSingleGraphSequencer
```

Fig. 2. Software architecture. In the main GNN folder, graph data representation classes are provided; `Models` sub-folder provides MLP, GNN, LGNN, CGNN (Composite GNN) and CLGNN (Composite Layered GNN) models implementations, while `Sequencers` sub-folder provides graph sequencers for feeding models with GraphObject/CompositeGraphObject data. Note that the aforementioned MLP model is a function that returns a Keras Sequential model, meaning that every Sequential model can be used for implementing net_s and net_o.

such cases, it is therefore possible to "homogenize" the different types of relationships, yielding a unique type of arc between nodes, described by a fixed-sized feature vector standing for the relationship it represents. Therefore, in the presented software, a heterogeneous or *composite* graph is described by its composite node feature matrix $\mathbf{X} \in \mathbb{R}^{n\times L_n}$, where $L_n = \max(L_n^K)$ – as a zero padding is added to shorter feature vectors – and by its arc feature matrix $\mathbf{E} \in \mathbb{R}^{m\times(2+l_e)}$. The incoming neighborhood set can be composed of nodes of different type $\mathrm{in}_v(v_i) = \{\mathrm{in}_v^k(v_i) \subseteq \mathrm{in}_v(v_i) \,:\, \mathrm{in}_v^k(v_i) \cap \mathrm{in}_v^h(v_i) = \emptyset,\ \forall k \neq h,\ \ k,h \in K\}$, where $\mathrm{in}_v^k(v_i) = \{v_j^k \in V : e_{ji} \in E,\ k \in K\}$ has size $|\mathrm{in}_v^k(v_i)|$.

In addition to functions defined in Eq. (3), in this domain, a further aggregation function φ_{cavg} is considered, defined as the sum of the averages over the number and type of the neighbors:

$$\varphi_{cavg}(v_i) = \sum_{k\in K} \frac{1}{|\mathrm{in}_v^k(v_i)|} \sum_{j\in \mathrm{in}_v^k(v_i)} (\mathbf{s}_j^{t-1}, \mathbf{x}_j^k, \mathbf{e}_{ji}) \tag{4}$$

For this purpose, it is quite useful to define the composite adjacency matrix set, used by the CompositeGNN to get the incoming message of a node, $\mathbf{A}_K = \{\mathbf{A}_k \in \mathbb{R}^{n\times n}\}$, where $\mathbf{A}_k = \{a_{ij}^k \in \mathbb{R} : k \in K, 0 \le i \le n-1, 0 \le j \le n-1\}$ is the composite adjacency matrix of type k. In particular, if there exists an arc $(v_i^k, v_j) \in E$, connecting two generic nodes $v_i^k, v_j \in V$, then $a_{ij}^k \in \mathbb{R} \neq 0$, otherwise $a_{ij}^k = 0$.

2.2. Software architecture

GNNkeras has been implemented as a module using the Python3 programming language and it is based on NumPy, SciPy, and TensorFlow libraries. This package includes GNN models for node-based, edge-based, and graph-based applications, working in homogeneous and heterogeneous graph domains, both in inductive and transductive learning contexts. NumPy and SciPy provide efficient numerical routines for dense and sparse data, while TensorFlow and Keras provide a simple and smart way to define and manage models, as well as to simplify the learning and evaluation processes. Fig. 2 shows a graphical representation of the package directory organization.

2.3. Software functionalities

The software relies on a custom graph representation, which is implemented in `graph_class.py` and defined as a GraphObject instance. For speeding up the learning procedure, before feeding a GNN model, a GraphObject is converted in another custom graph representation, called GraphTensor, which contains a tensor-based description of all the attributes for the graph to be correctly and quickly processed by the GNN model. In the heterogeneous setting, another class defined by CompositeGraphObject/CompositeGraphTensor is provided in `composite_graph_class.py`. GraphObject and GraphTensor – as well as their heterogeneous versions – are the data types GNNkeras is based on. Although they represent the same object, they differ in the data types used for their attributes: GraphObject is described by NumPy arrays and SciPy sparse matrices while GraphTensor – as the name suggests – by TensorFlow constant and sparse tensors.

2.4. Graph data type

An instance of GraphObject/GraphTensor is therefore a compact representation of a generic graph $G = (V, E)$, which is initialized at least by a node feature matrix $\mathbf{X}$, an arc feature matrix $\mathbf{E}$, and by a target matrix $\mathbf{T}$. Since not all $v_i \in V$ or $e_{ij} \in E$ are necessarily associated with a target value, a boolean output mask $\mathbf{m}_o \in \mathbb{B}$ is included in the GraphObject, to define whether or not a target value $t_i \in \mathbf{T}$ is associated with a specific node or arc. Moreover, when the dataset is composed of only one single graph, a boolean set mask $\mathbf{m}_s \in \mathbb{B}$ is included, so as to specify the subset of nodes or arcs belonging to a specific dataset or data batch. Note that, for the graph to be correctly processed, the dimensions of $\mathbf{m}_s$ and $\mathbf{m}_o$ must match, while $\mathbf{m}_o$ must contain as many true values as the number of values in $\mathbf{T}$. In the heterogeneous domain, an instance of CompositeGraphObject or CompositeGraphTensor includes a boolean type mask $\mathbf{m}_K \in \mathbb{B}^{n\times K} = \{m_{ik}^K \in \mathbb{B}\}$, to specify the type for each node, such that $m_{ik}^K = 1$ if and only if node $v_i = v_i^k$, otherwise $m_{ik}^K = 0$. During the initialization phase, a GraphObject defines automatically three SciPy sparse matrices in coordinate format: the adjacency matrix $\mathbf{A}$ and an arc–node matrix $\mathbf{A_N}$, which are used in the aggregation message procedure in the state transition phase – affected by a hyperparameter `aggregation_mode`, whose value defines the φ version to be used for aggregating the messages –, and the node-graph matrix $\mathbf{N_G}$, which is used by the GNN models in graph-based applications to convert a node-based output to an overall graph-based output, calculated as the average of the nodes' output. A GraphObject and a CompositeGraphObject can be also saved in a single NumPy uncompressed/compressed npz file – or in a folder of text files – which includes all the necessary matrices for their complete representation. Given a dataset of graphs, in the form of a list of graph data elements, these classes also provide a smart way to save the entire dataset in a single folder, from which it can be loaded when needed.

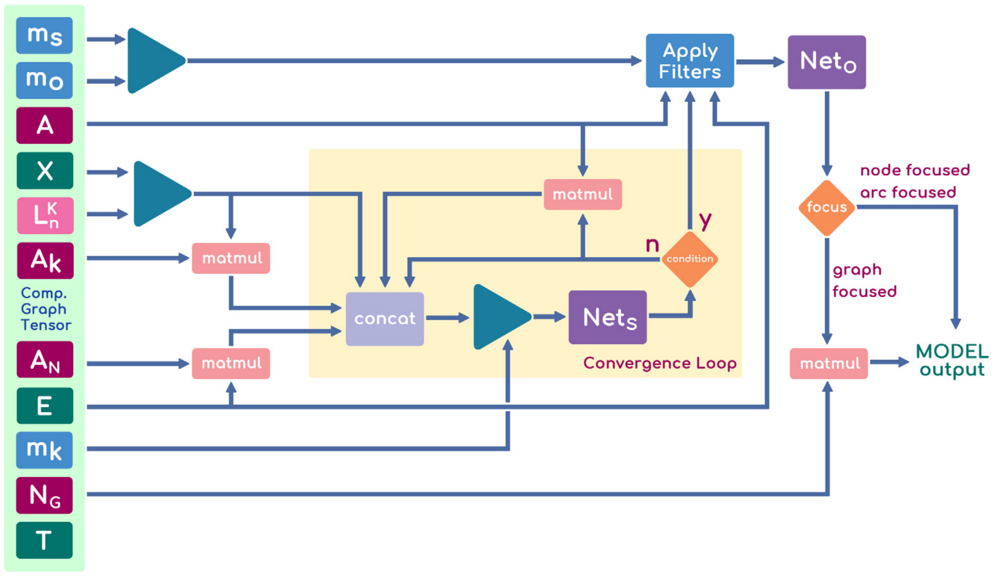


Fig. 3. Composite GNN model. The GraphSequencer generates from GraphObjects batches of GraphTensor which are presented to the model as input. All quantities pass through multiple operations (matrix multiplications, boolean mask filtering, and concatenating processes) to form the input to net_s and net_o.

In order to be correctly processed by the GNN models, GraphObjects and CompositeGraphObjects are required to be fed to a special data handler, the Graph Sequencer, described in the following.

2.5. Graph Sequencer

A GraphSequencer is a data manager for fitting to a sequence of data – such as a dataset of graphs – which is fed with a single GraphObject or a list of GraphObject elements to generate batches of GraphTensors, whose attributes are presented as input to the given GNNkeras model. A total of six GraphSequencer are provided, for multi-graph and single-graph-based datasets, in homogeneous and heterogeneous graph domains, and for inductive and transductive learning approaches. It is worth noting that the transductive one is a special class of Sequencer, which is fed with homogeneous GraphObjects while generating heterogeneous graph data. Indeed, for each epoch and batch, it splits the graph training targeted nodes into two subsets of inductive and transductive nodes. In the transductive set, the target of a node is integrated in its feature vector. Therefore, for that node, no supervision is used in the learning process. Instead, no operation is carried out on the (supervised) inductive set, which is used in the learning process for adapting the model's parameters. As a consequence, the new graph cannot be represented by a homogeneous GraphTensor, since two types of nodes are present – described by feature vectors of different lengths – thus making necessary the CompositeGraphTensor representation.

2.6. GNN models

Since GNNkeras is a Keras-based software, where the GNN classes inherit from the `Keras.Model` class, it comes with all the functionalities provided by TensorFlow 2.x and Keras (TensorFlow backend). To parallelize software execution on modern CPUs and GPUs, all the operations have been based on matrix multiplications. Fig. 3 shows the processing scheme of a heterogeneous graph by a Composite GNN model.

3. Illustrative examples

An example of the software application in a homogeneous graph-domain problem is presented below. Let $\mathcal{G}$ be a dataset of N graphs, with $N > 100$, described by three lists of $\mathbf{X}$, $\mathbf{E}$, $\mathbf{T}$ matrices of equal length – `nodes`, `arcs`, `targs` lists – such that the i-th element refers to the i-th graph. Let net_s and net_o be described by MLP `Keras.Sequential` models. Then `net_state` and `net_output` refer to two single models, net_s and net_o, for a single GNN model. In the LGNN case, `nets_state` and `nets_output` refer to two lists having length equal to the number of layers in the model and containing net_s and net_o instances for each GNN layer of the LGNN model. Before defining the GNN as well as the LGNN model, let set some parameters:

```
# GNN
state_dimension = 3
max_iteration = 5
state_threshold = 0.01

# LGNN
get_state = False
get_output = True
training_mode = 'serial'

# Learning procedure
loss_function = tf.Keras.losses.categorical_crossentropy
optimizer = tf.optimizers.Adam(learning_rate=0.001)
epochs = 10

# Graphs
aggregation_mode = 'average'  # incoming message policy
addressed_problem = 'c'     # for classification problems
focus = 'g'        # for graph-based problems
```

Each element of `nodes`, `arcs` and `targs` is necessary for the construction of the single `GraphObject` instance. Once the dataset is built, it is fed to two `MultiGraphSequencer` – since the dataset is composed of multiple graphs – for the training and the test set, respectively. In the following, only the last 100 graphs are fed to the test Sequencer, while the others are used for feeding the training Sequencer.

```
# Dataset
graphs = [GraphObject(nodes=n, arcs=e, targets=t,
            focus=focus,
            aggregation_mode=aggregation_mode)
      for e, n, t in zip(arcs, nodes, targs)]
```

```
# Sequencer for training and test set
gTr_Sequencer = MultiGraphSequencer(graphs[:-100],
                                    focus,
                                    aggregation_mode)

gTe_Sequencer = MultiGraphSequencer(graphs[-100:],
                                    focus,
                                    aggregation_mode)
```

Since the GNN and LGNN models belong to the `Keras.Model` class, they need to be compiled: for both models, the parameter `run_eagerly=True` should be set, as the TensorFlow graph mode is not available in this software version. The LGNN model can be compiled with a special parameter `training_mode`, which affects its learning behavior. In particular, it can be set to `serial`, `residual` or `parallel`. In `serial` mode, each GNN layer is trained separately, one by one, as described in the original paper [16]; in `parallel` mode, GNN layers are trained simultaneously, by considering a loss which is the sum of the losses between the target and the output of each GNN layer, and backpropagating the error throughout all the GNN layers; finally, `residual` mode allows to train the GNN layers simultaneously, by considering the loss between the target and the sum of the outputs of all GNNs, and backpropagating the error throughout all the GNN layers.

```
# GNN model
gnn = GNNgraphBased(net_state=net_state, net_output=net_output,
            state_vect_dim=state_dimension,
            max_iteration=max_iteration,
            state_threshold=state_threshold)

gnn.compile(optimizer=optimizer, loss=loss_function, run_eagerly=True)

# LGNN model
gnnLayers = [GNNgraphBased(net_state=s, net_output=o,
                state_vect_dim=state_dimension,
                max_iteration=max_iteration,
                state_threshold=state_threshold)
        for s, o in zip(nets_state, nets_output)]

lgnn = LGNN(gnns=gnnLayers, get_state=get_state, get_output=get_output)
lgnn.compile(optimizer=optimizer, loss=loss_function,
        run_eagerly=True, training_mode=training_mode)
```

The models are defined and compiled: the learning and evaluation procedures can take place. For illustrative purposes only, no validation data or TensorFlow callbacks are provided during the training process.

```
### Learning procedure
gnn.fit(gTr_Sequencer, epochs=epochs)
lgnn.fit(gTr_Sequencer, epochs=epochs)

### Evaluation procedure
gnn.evaluate(gTe_Sequencer)
lgnn.evaluate(gTe_Sequencer)
```

4. Impact

Graph data are nowadays ubiquitous, allowing to represent relational information between data entities in many different research domains. Furthermore, the importance of graphs is increasing as they permit to directly merge information from different sources, which is very natural in modern applications. On the one hand, graphs are a powerful form of data representation, in particular for relational information, and applications of machine learning techniques in this domain become more important every day. On the other hand, standard machine learning methods were based on flat vectorial data, thus their application on graph domains was difficult. This implied that graph data had to be preprocessed, losing relevant pieces of information, in particular on the relational side. This limit stimulated the proposal and the success of GNNs, which are a class of machine learning models that can process graph data directly. In the last few years, the interest of researchers in the field has known a recent and steady increase, leading to the development of a large number of new models [15], several theoretical studies [20,21], so as a huge number of real-world applications [13,22].

GNNkeras has been designed in this context with the aim of simplifying the use of the GNNs. Graph-based networks can be classified into two broad classes, recurrent and convolutional [15]. GNNkeras is focused on the former class of GNNs, on which our group has accumulated long expertise. As far as we know, this software is the first solution for TensorFlow 2.x specifically designed for recurrent GNNs. The main difference between these two kinds of models is the way they operate directly on graphs. In the Convolutional case, the graph is processed by means of convolutional and pooling techniques, which are carried out by a fixed number of stacked layers, using independent parameters in the various layers. Instead, in the recurrent case, the MLP units share the architecture and parameters, and the state computation can be carried out for a fixed number of times or until convergence to a steady state.

The characteristics of GNNkeras are many and can be summarized in the following points.

- GNNkeras allows to develop and deploy GNN models easily, in a few lines of code, and with high versatility. Representing a GNN as a GNNkeras model gives a considerable advantage compared to previous common solutions, which were manually written from scratch with TensorFlow.
- All the three different types of deep learning problems on graphs are implemented: node-based, edge-based, graph-based.
- GNNs can be layered, implementing the LGNN version for more complex problems.
- GNNs and LGNNs can be applied to heterogeneous graphs.
- All the three super-categories of deep learning tasks can be tackled with GNNs: regression, classification, and generation.
- Inductive and mixed inductive–transductive learning can be adopted.

The expected impact of GNNkeras is mainly related to its capability of helping its users in speeding up the proposal of new research and the development of advanced software. We think that, due to the mentioned characteristics, GNNkeras is a flexible and suitable tool to exploit ML for graph data. The library can be used by researchers in ML to test new models and to design new applications. It can be also used by software developers from companies and organizations designing applications for relational data. Finally, it is worth noting that the exceptional interest in ML for graphs is a measure of the size and growth of the community operating in the sector and for which GNNkeras can be useful.

5. Conclusions

In this paper, a new general GNN framework has been presented, which provides multiple Keras-based GNN models for homogeneous and heterogeneous graph processing for both inductive and transductive learning approaches. It was developed to help the research and the development of software applications in the field of ML for graphs, to simplify the approach to this kind of machine learning models for those who are already familiar with Keras models as well as for those who want to enter the graph domain of Artificial Intelligence.

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.

References

[1] Grindley Helen M, Artymiuk Peter J, Rice David W, Willett Peter. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. J Mol Biol 1993;229(3):707–21.

[2] Gardiner Eleanor J, Artymiuk Peter J, Willett Peter. Clique–detection algorithms for matching three–dimensional molecular structures. J Mol Graph Model 1997;15(4):245–53. http://dx.doi.org/10.1016/S1093-3263(97)00089-2.

[3] Pancino Niccolò, Rossi Alberto, Ciano Giorgio, Giacomini Giorgia, Bonechi Simone, Andreini Paolo, Scarselli Franco, Bianchini Monica, Bongini Pietro. Graph Neural Networks for the Prediction of Protein–Protein Interfaces. In: Proceedings of the 28th european symposium on artificial neural networks, computational intelligence and machine learning, ESANN 2020, Bruges, Belgium, October 2–4; 2020, pp. 127–32.

[4] Zitnik Marinka, Agrawal Monica, Leskovec Jure. Modeling polypharmacy side effects with graph convolutional networks. Bioinformatics 2018;34(13):i457–66.

[5] Scarselli Franco, Gori Marco, Tsoi Ah-Chung, Hagenbuchner Markus, Monfardini Gabriele. The graph neural network model. IEEE Trans Neural Netw 2009;20:61–80.

[6] Kipf Thomas N, Welling Max. Semi–supervised classification with graph convolutional networks. 2016, arXiv preprint arXiv:1609.02907.

[7] Hamilton William L, Ying Rex, Leskovec Jure. Inductive representation learning on large graphs. In: Proceedings of the 31st International Conference on Neural Information Processing Systems; 2017, pp. 1025–35.

[8] Veličković Petar, Cucurull Guillem, Casanova Arantxa, Romero Adriana, Liò Pietro, Bengio Yoshua. Graph attention networks. 2017, arXiv preprint arXiv:1710.10903.

[9] Battaglia Peter W, Hamrick Jessica B, Bapst Victor, Sanchez-Gonzalez Alvaro, Zambaldi Vinicius, Malinowski Mateusz, Tacchetti Andrea, Raposo David, Santoro Adam, Faulkner Ryan, Gulcehre Caglar, Song Francis, Ballard Andrew, Gilmer Justin, Dahl George, Vaswani Ashish, Allen Kelsey, Nash Charles, Langston Victoria, Dyer Chris, Heess Nicolas, Wierstra Daan, Kohli Pushmeet, Botvinick Matt, Vinyals Oriol, Li Yunjia, Pascanu Razvan. Relational inductive biases, deep learning, and graph networks. 2018, arXiv preprint arXiv:1806.01261.

[10] Hsieh Kanglin, Wang Yinyin, Chen Luyao, Zhao Zhongming, Savitz Sean, Jiang Xiaoqian, Tang Jing, Kim Yejin. Drug repurposing for covid–19 using graph neural network and harmonizing multiple evidence. Sci Rep 2021;11:23179.

[11] Sanchez-Gonzalez Alvaro, Godwin Jonathan, Pfaff Tobias, Ying Rex, Leskovec Jure, Battaglia Peter. Learning to simulate complex physics with graph networks. In: Daumé Hal, Singh Aarti, editors. Proceedings of the 37th international conference on machine learning. Proceedings of machine learning research, 119, PMLR; 2020, p. 8459–68.

[12] Ying Rex, He Ruining, Chen Kaifeng, Eksombatchai Pong, Hamilton William L, Leskovec Jure. Graph convolutional neural networks for web–scale recommender systems. In: Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining; 2018, pp. 974–83.

[13] Zhou Jie, Cui Ganqu, Hu Shengding, Zhang Zhengyan, Yang Cheng, Liu Zhiyuan, Wang Lifeng, Li Changcheng, Sun Maosong. Graph neural networks: A review of methods and applications. AI Open 2020;1:57–81.

[14] Bacciu Davide, Errica Federico, Micheli Alessio, Podda Marco. A gentle introduction to deep learning for graphs. Neural Netw 2020;129:203–21.

[15] Wu Zonghan, Pan Shirui, Chen Fengwen, Long Guodong, Zhang Chengqi, Yu Philip S. A comprehensive survey on graph neural networks. IEEE Trans Neural Netw Learn Syst 2021;32(1):4–24. http://dx.doi.org/10.1109/TNNLS.2020.2978386.

[16] Bandinelli Niccolo, Bianchini Monica, Scarselli Franco. Learning long–term dependencies using layered graph neural networks. In: The 2010 International Joint Conference on Neural Networks (IJCNN). IEEE; 2010, p. 1–8.

[17] Rossi Alberto, Tiezzi Matteo, Dimitri Giovanna Maria, Bianchini Monica, Maggini Marco, Scarselli Franco. Inductive–transductive learning with graph neural networks. In: IAPR Workshop on Artificial Neural Networks in Pattern Recognition. Springer; 2018, p. 201–12.

[18] Ciano Giorgio, Rossi Alberto, Bianchini Monica, Scarselli Franco. On inductive–transductive learning with graph neural networks. IEEE Trans Pattern Anal Mach Intell 2022;44(2):758–69. http://dx.doi.org/10.1109/TPAMI.2021.3054304.

[19] Abadi Martín, Agarwal Ashish, Barham Paul, Brevdo Eugene, Chen Zhifeng, Citro Craig, Corrado Greg S, Davis Andy, Dean Jeffrey, Devin Matthieu, Ghemawat Sanjay, Goodfellow Ian, Harp Andrew, Irving Geoffrey, Isard Michael, Jia Yangqing, Jozefowicz Rafal, Kaiser Lukasz, Kudlur Manjunath, Levenberg Josh, Mané Dandelion, Monga Rajat, Moore Sherry, Murray Derek, Olah Chris, Schuster Mike, Shlens Jonathon, Steiner Benoit, Sutskever Ilya, Talwar Kunal, Tucker Paul, Vanhoucke Vincent, Vasudevan Vijay, Viégas Fernanda, Vinyals Oriol, Warden Pete, Wattenberg Martin, Wicke Martin, Yu Yuan, Zheng Xiaoqiang. TensorFlow: Large-Scale Machine learning on heterogeneous systems, Software available from tensorflow.org. 2015, https://www.tensorflow.org/.

[20] Sato Ryoma. A survey on the expressive power of graph neural networks. 2020, arXiv preprint arXiv:2003.04078.

[21] D'Inverno Giuseppe Alessio, Bianchini Monica, Sampoli Maria Lucia, Scarselli Franco. An unifying point of view on expressive power of GNNs. 2021, arXiv preprint arXiv:2106.08992.

[22] Bongini Pietro, Bianchini Monica, Scarselli Franco. Molecular generative graph neural networks for drug discovery. Neurocomputing 2021;450:242–52.