IGNNITION: A FRAMEWORK FOR FAST PROTOTYPING OF GRAPH NEURAL NETWORKS

David Pujol-Perich [1] **José Suárez-Varela** [1] **Miquel Ferriol-Galmés** [1] **Shihan Xiao** [2] **Bo Wu** [2]
Albert Cabellos-Aparicio [1] **Pere Barlet-Ros** [1]

ABSTRACT

Recent years have seen the vast potential of *Graph Neural Networks* (GNN) in many fields where data is structured as graphs (e.g., chemistry, logistics). However, implementing a GNN prototype is still a cumbersome task that requires strong skills in neural network programming. This poses an important barrier to researchers and practitioners that want to apply GNN to their specific problems but do not have the needed Machine Learning expertise. In this paper, we present IGNNITION, a novel open-source framework for fast prototyping of GNNs. This framework is built on top of TensorFlow, and offers an intuitive high-level abstraction that allows the user to define its GNN model via a YAML file, being completely oblivious to the tensor-wise operations made internally by the model. At the same time, IGNNITION offers great flexibility to build any GNN-based architecture. To showcase its versatility, we implement two state-of-the-art GNN models applied to the field of computer networks, which differ considerably from well-known standard GNN architectures. Our evaluation results show that the GNNs produced by IGNNITION are equivalent in performance to implementations directly coded in TensorFlow.

1 INTRODUCTION

Graph Neural Networks (GNN) (Scarselli et al., 2008) have recently become a hot topic among the Machine Learning (ML) community. The main novelty behind GNNs is their unique ability to learn and generalize over graph-structured information. This has enabled the development of many GNN-based applications in different fields where data is fundamentally represented as graphs – E.g., chemistry (Gilmer et al., 2017; You et al., 2018), physics (Battaglia et al., 2016; Farrell et al., 2018), biology (Zitnik et al., 2018; Gainza et al., 2020), information science (Ying et al., 2018).

Nowadays, designing and implementing a GNN-based solution involves dealing with complex mathematical formulations and programming with tensor-oriented ML libraries, such as TensorFlow (Abadi et al., 2016) or PyTorch (Paszke et al., 2019). At the same time, applying GNN to specific problems (e.g., computer networks, chemistry) often requires the design of ad-hoc GNN architectures adapted to the data under consideration – E.g., heterogeneous graphs (Rusek et al., 2019; Geyer and Carle, 2018; Badia-Sampera et al., 2019). This represents a critical entry barrier for researchers and practitioners from different fields that could benefit from the use of GNN, but lack the necessary ML expertise to implement models tailored to their needs.

In this paper, we present IGNNITION, a TensorFlow-based framework for fast prototyping of GNNs. This framework is open source[1], and mainly targets users with little background on neural network programming. With IGNNITION, users can easily design their own GNN models – including complex non-standard architectures – via an intuitive, human-readable YAML file. Based on this input, the framework automatically generates an efficient implementation of the GNN in TensorFlow, making the user completely oblivious to the underlying tensor-wise operations.

To achieve this, we propose a high-level abstraction called the *Multi-Stage Message Passing graph* (MSMP graph). This novel abstraction covers a broad definition of GNN, which provides great flexibility to design variants and combinations of state-of-the-art GNN architectures – E.g., Message Passing Neural Networks (Gilmer et al., 2017), Graph Convolutional Networks (Kipf and Welling, 2017), Gated Neural Networks (Li et al., 2016), Graph Attention Networks (Veličković et al., 2017), Graph LSTM (Liang et al., 2016), Typed Graph Networks (Prates et al., 2019). Likewise, MSMP graphs enable to hide the complex mathematical formulation behind the implementation of a GNN.

[1]Barcelona Neural Networking Center, Universitat Politècnica de Catalunya, Spain [2]Network Technology Lab., Huawei Technologies Co.,Ltd.. Correspondence to: David Pujol-Perich <david.pujol.perich@upc.edu>.

Proceedings of the First MLSys Workshop on Graph Neural Networks and Systems (GNNSys'21), San Jose, CA, USA, 2021.

[1]Available at: https://ignnition.net

In contrast, existing ML libraries with support for GNN (You et al., 2020; Battaglia et al., 2018; Fey and Lenssen, 2019; Wang et al., 2019; Grattarola and Alippi, 2020) are either considerably more complex or lack sufficient flexibility to implement non-standard GNN models. For instance, GraphGym (You et al., 2020) does not support complex message passing strategies used in well-known GNN models – e.g., multi-step message passing (Rusek et al., 2019; Geyer and Carle, 2018); while other approaches, such as DGL (Wang et al., 2019) or Graph Nets (Battaglia et al., 2018), still require a high degree of expertise in neural network programming compared to IGNNITION.

More in detail, the main features of IGNNITION are:

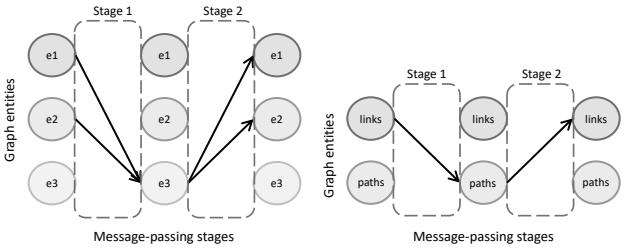
- **High-level abstraction:** It introduces a human-readable interface that abstracts away the mathematical formulation behind GNN implementations.
- **Flexibility of design:** Support for any type of GNN and message passing scheme via the novel MSMP graph abstraction.
- **High performance:** It produces efficient GNN implementations, equivalent to native code, as later shown in the evaluation.
- **Easy debugging:** It incorporates interactive debugging visualizations and advanced error-checking mechanisms to help users troubleshoot their GNN models.

With IGNNITION, users with little experience on neural network programming (e.g., TensorFlow, PyTorch) can reduce significantly the time needed to achieve functional GNN implementations adapted to their problems. To show the versatility of this framework, in this paper we use IGNNITION to implement two state-of-the-art GNN models applied to computer networks. As discussed below, these models include several particularities that differ considerably from standard GNN architectures.

2 The Multi-Stage Message Passing Graph Abstraction

This section introduces a novel high-level abstraction we called the *Multi-Stage Message Passing graph* (hereafter MSMP graph).

The MSMP graph abstraction provides an interface with a flexible modular design, offering support for any variant of state-of-the-art GNN architectures as well as custom combinations of individual components of them (e.g., message, aggregation, update, loss, normalization functions). Hereafter, we refer to the message-passing phase as the pipeline of message-aggregation-update layers that shape a GNN (Battaglia et al., 2018).



(a) MSMP Example (b) RouteNet's MSMP

Figure 1: MSMP representations – Generic example and RouteNet (Rusek et al., 2019)

One main novelty of this abstraction is that it offers support for non-standard message-passing schemes divided in multiple stages and including different element types – e.g., hypergraphs (Feng et al., 2019), heterogeneous graphs (Rusek et al., 2019). This enables to implement a wide spectrum of GNN proposals applied to different areas of knowledge (Zhou et al., 2018).

Particularly, with the MSMP graph, a GNN can be intuitively defined by a set of graph entities and how they relate to each other, which eventually describes a message-passing iteration of the GNN. Fig. 1a illustrates an example of a GNN with three different entity types ($e1$, $e2$ and $e3$). In this MSMP graph, we can observe two differentiated stages in the message-passing phase. In the first stage, elements of entity $e1$ and $e2$ share their hidden states with their neighbors of entity $e3$ according to the connections of the input graph. Then, in the second stage, $e3$ elements share their states with their linked elements of type $e1$ and $e2$. This process is then repeated a number of iterations T or until a convergence criterion is satisfied (Scarselli et al., 2008).

As a result, IGNNITION supports any GNN that can be represented as an MSMP graph. This broadly includes the main state-of-the-art GNN architectures and many possible variants, such as Graph Attention Networks (Veličković et al., 2017), Graph Convolutional Networks (Kipf and Welling, 2017), Gated Neural Networks (Li et al., 2016), Graph LSTM (Liang et al., 2016), Typed Graph Networks (Prates et al., 2019), Hypergraph Neural Networks (Feng et al., 2019), and many others.

3 Framework Implementation

This section describes the four main modules that shape IGNNITION. These are: the Model Description Interface (Sec. 3.1), the Dataset Interface (Sec. 3.2), the Core Engine (Sec. 3.3), and the Debugging Assistant (sec. 3.4).

3.1 Model Description Interface

The first step to build a GNN with IGNNITION is to describe the model using the MSMP graph abstraction pre-

```
entities:
- name: path
  state_dimension: 32
  initial_state:
  - type: build_state
    input: [traffic]
```

Figure 2: Entity definition in the YAML model description.

sented in Section 2. This can be done by filling a small YAML file. The description should include all the entities involved and the relationships between them. Thus, in IGNNITION the GNN model definition remains completely decoupled from the input dataset and some other implementation details. More specifically, this YAML file should contain the following information:

3.1.1 *Entities definition*

First of all, the user defines the different entities to consider in the problem scenario – and consequently in the corresponding MSMP graph. Particularly, the user should first indicate the entity names, the size of the state vectors, and how to initialize these state vectors. For instance, in the context of computer networks, an entity typically represents a set of network components, which can be physical (e.g., routers, links), or logical (e.g., paths, virtual network functions). As an example, Figure 2 shows the definition of an entity (*path*) in the YAML model description file. The state initialization is defined by a flexible pipeline of operations that supports complex initialization methods used in state-of-the-art models (e.g., NN-based feature embedding). Note that these operations reference features through unique names (e.g., "traffic") that are then used to identify them in the input dataset – as shown later in Sec. 3.2. Optionally, a normalization function can be applied to features before they are introduced into the initial state vectors (e.g., "z-score").

3.1.2 *Message passing phase*

Then, the user needs to complete the MSMP graph by defining the message-passing operations in the GNN model – i.e., the relations between entities. To do so, it is required to describe a single message-passing iteration, which can be divided in turn in several stages. In each message-passing stage, a set of entities share their hidden states with some other entities. For example, in the MSMP graph previously presented in Figure 1a the message passing is divided in two stages. In the first one, the following message passings are executed: $e1 \rightarrow e3$, and $e2 \rightarrow e3$; while in the second stage the reverse message passings are performed: $e3 \rightarrow e1$, and $e3 \rightarrow e2$. Once a complete message passing iteration is defined, IGNNITION is able to automatically generate a GNN model that unrolls this message passing scheme a number of iterations (T) defined by the user.

With IGNNITION, users can intuitively define the message passing by specifying in the model description file a set of YAML keywords describing each stage. A full description with all the available keywords can be found at (BNN Center, 2021). To do so, the user must first specify the names of the source and destination entities. Note that there may be more than one entity type that sends its states to the destination elements. For instance, in stage #1 of the MSMP graph in Fig. 1a, $e1$ and $e2$ entities send simultaneously their hidden states to nodes of type $e3$. The adjacencies between elements of the source and destination entities are completely determined by the graphs of the input dataset. Finally, the user needs to define the *message*, *aggregation* and *update* functions used in each message passing. To this end, IGNNITION provides full flexibility to implement these functions in many different ways (e.g., neural networks, element-wise multiplications or even direct assignments) that are often used in state-of-the-art GNN models (Zhou et al., 2018).

```
- destination_entity: link
  source_entities:
    - name: path
      message:
            - type: direct_assignment

  aggregation:
        - type: sum

  update:
        type: neural_network
        nn_name: recurrent_1
```

Figure 3: Example of a message passing definition between two entities (*paths* to *links*) in RouteNet (Rusek et al., 2019).

As an illustrative example, Figure 3 shows the definition of stage #2 in the message passing of RouteNet (Rusek et al., 2019) – see Fig. 1b. To define the message passing (from "path" to "link" entities) we first set the general information of the message passing (source/destination entity names). Then, we define the message, aggregation and update functions with a few lines of YAML text. To achieve a modular design, in case of using NNs (e.g., in the *update* function), the user can define a name (e.g., "recurrent_1") that is then used to describe the NN details in a separate section of the model description file.

3.1.3 *Readout*

At this point, the user is asked to define the *readout* function. Note that state-of-the-art GNN models often chain multiple operations to implement the readout (Geyer and Carle, 2018). Thus, in IGNNITION the readout can be defined via a flexible pipeline of instructions (e.g., *pooling*, *neural_network*). Likewise, it can finally produce per-node or global graph-level outputs.

Particularly, the definition must contain a set of YAML objects –described in detail in (BNN Center, 2021). For

```
- nn_name: readout_model
  nn_architecture:
  - type: Dense
    units: 256
    kernel_regularizer: 0.1
    activation: selu

  - type: Dropout
    rate: 0.5

  - type: Dense
    units: 1
```

Figure 4: Example of the definition of a neural network.

each operation, the user needs to define the type (e.g., "pooling", "neural_network") and the names of the *input* elements. These can refer to the output of previous operations or to the final hidden states of a certain entity. Similarly to the message-passing functions, a name can be used to then describe separately the NN architecture used for these operations. Likewise, the last operation of the readout must contain the name of the output label, which is then used to reference it in the training dataset.

3.1.4 Neural networks definition

Finally, the user must define in a separate section all the neural networks that were previously referenced (e.g., for the message, update and readout functions). For this purpose, IGNNITION offers an interface that maps directly to native functions of the well-known *Keras* library (Chollet et al., 2015). Thus, to define for instance a feed-forward neural network, the user can specify each layer separately, indicating the layer's type and any additional Keras parameters (see Fig. 4). Alternatively, the user can instantiate any other NN models such as Recurrent Neural Networks (RNN) by using their respective Keras parameters. To facilitate this process, a debugging agent (Sec. 3.4) assists users in case they do not define these NNs correctly.

As we can observe in the examples of this section, with IGNNITION the user can define a GNN model – via MSMP graphs – while remaining completely oblivious to the complex tensor-wise operations behind the resulting TensorFlow implementation produced by this framework.

3.2 Dataset Interface

Another main module of the framework is the *Dataset interface*. This interface permits to decouple the GNN model description from the input data. Likewise, it enables to easily feed the model with datasets in heterogeneous formats. To achieve this, IGNNITION provides an interface that enables to process datasets with the well-known *NetworkX* library (Hagberg et al., 2008). This library already implements a plethora of functions that automatize the definition of graphs from datasets, as well as serializing the resulting graphs to standard formats. Particularly, IGNNITION reads any dataset serialized by NetworkX in JSON format. Note

```
// main.py
import ignnition

def main():
    model = ignnition.create_model(model_dir=<PATH>)
    model.train_and_validate()
```

Figure 5: Python code to generate and train a GNN model.

that datasets can contain graphs of different sizes and structures (e.g., data from different networks). In this regard, the GNN implementations generated by IGNNITION assemble at runtime the message-passing functions according to the elements and connections of input graphs.

3.3 Core Engine

This module contains the main logic behind IGNNITION. To this end, it implements internally the MSMP graph abstraction proposed in Section 2.

Once the model has been designed (Sec. 3.1) and the dataset has been properly formatted (Sec. 3.2), the user can interact with the *Core Engine* to execute the framework's functionalities – E.g., train and validate a GNN model, make predictions, obtain visual representations of the model (Sec. 3.4). Note that two API calls are enough to execute any of these functionalities – as shown in the example of Fig. 5.

As mentioned previously, IGNNITION implements the designed GNN model in TensorFlow. This enables to work internally with the efficient computational graph generated by this ML library. As a result, the framework is able to achieve comparable performance to implementations directly coded in TensorFlow – as shown later in Section 4.

To create the model implementation, the Core Engine proceeds in a similar way to traditional compilers, following three main steps to parse the model description file: (i) lexical analysis, (ii) syntactical analysis and (iii) semantic analysis. These three steps enable to detect unexpected structures and trigger numerous error-checking mechanisms (Sec. 3.4).

After validating all the input information, IGNNITION automatically generates the GNN model implementation. For this, it makes use of a generic definition of MSMP graphs that covers a broad spectrum of particularities introduced by the user in its model definition.

3.4 Debugging Assistant

One of the biggest challenges when designing NN models with traditional ML libraries is to find potential bugs and fix them. Having a clear picture of the resulting GNN implementation is often a non-trivial and time-consuming task. For this reason, IGNNITION incorporates an advanced debugging system that assists users in different ways.

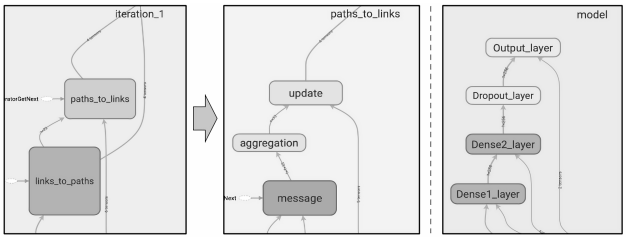


Figure 6: Debugging visualizations — One Message Passing (MP) iteration (left), paths_to_links MP stage (middle), and readout (right) of RouteNet (Rusek et al., 2019).

First, it automatically produces interactive visual representations of the internal GNN architecture. To do so, it relies on a Tensorboard-based system that enriches the visualization of the model at different levels of granularity. For example, Figure 6 shows three screenshots obtained after implementing RouteNet (Rusek et al., 2019) with IGNNITION. Particularly, we can observe the two message-passing stages of this model (i.e., *links_to_paths* and *paths_to_links*), and a zoomed-in visualization of the second stage (*paths_to_links*) with its internal functions (*message*, *aggregation*, and *update*). The image on the right shows the neural network layers that compose the readout function. Additionally, numerous advanced error-checking mechanisms are incorporated to ensure the correct definition of the model and to provide guidance to fix potential badly defined fields (e.g., wrong NN descriptions, entities definitions, missing features in datasets).

4 EMPIRICAL EVALUATION

This section presents several paradigmatic use cases where we leverage IGNNITION to make fast GNN implementations. In particular, we aim to showcase the flexibility of this framework to implement complex non-standard GNN models. We intentionally select two models applied to computer networks due to their particularly complex message-passing architectures. Likewise, we compare the performance of the implementations produced by IGNNITION and their equivalent implementations in TensorFlow trough two main metrics: (i) the *accuracy* achieved by the models after training, and (ii) the *execution cost* of both implementations.

4.1 GNN models description

We show below a brief description of the two implemented models and their main architectural singularities:

4.1.1 RouteNet

RouteNet (Rusek et al., 2019) was proposed as a network modeling tool that predicts per-path performance metrics (e.g., delay, jitter) given a network snapshot as input, defined by: a network topology, a routing configuration, and a traffic matrix. To this end, this GNN architecture considers heterogeneous graphs with two different entity types (*links* and *paths*) and a two-stage message passing scheme – previously illustrated in the MSMP graph of Fig. 1b.

4.1.2 Graph-Query Neural Network (GQNN)

The GQNN model (Geyer and Carle, 2018) addresses a different problem: supervised learning of traditional network routing protocols with GNN, such as shortest path or max-min routing. To this end, this GNN model uses a novel architecture with two entity types: *routers* and *interfaces*, being the latter the several network interfaces of each router in the network. This model considers a single-stage message passing scheme where routers and interfaces share their hidden states. As output, the model determines whether the interfaces transmit traffic or not (i.e., [0,1]), which eventually defines the routing configuration of the network. Another particularity of this model is in the readout, which uses an operation pipeline with an element-wise multiplication and then a NN for the final prediction.

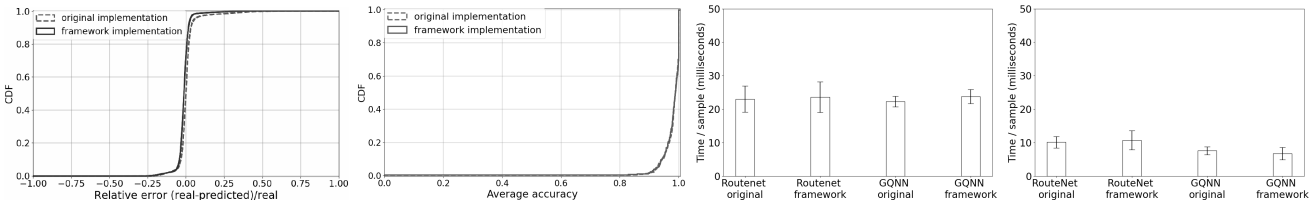
4.2 Evaluation: Accuracy and Cost

To evaluate the performance of such models, we use their native implementations in TensorFlow as a reference, and reproduce some of the experiments made in their corresponding papers. Particularly, we use two datasets with *300,000* and *40,000* samples originally used in the evaluations of RouteNet and GQNN respectively. In each case, we randomly select 80% of the samples for training and 20% of the samples for evaluation.

To evaluate the accuracy of the models, we train them under equal conditions. Figure 7a shows the Cumulative Distribution Function (CDF) of the relative error produced by the IGNNITION's implementation of RouteNet and its corresponding native Tensorflow implementation. Figure 7b, similarly shows the CDF of the classification accuracy achieved by the GQNN implementations. As we can observe, in both experiments IGNNITION implementations achieve equivalent accuracy after being trained with the same samples, which proves a consistent behavior of this framework across different GNN architectures.

Likewise, we evaluate the execution cost of both classes of implementations. Figure 7c depicts the average execution time per sample during training; while Figure 7d shows the same results for inference. In line with the previous results, we observe that the execution cost of IGNNITION implementations is equivalent to that of the original models in TensorFlow, both for training and inference. Note that all these experiments were made in a controlled environment, using the same computing resources for all the cases.

As a conclusion, we can observe that the resulting imple-



(a) RouteNet (b) GQNN (c) Training time per sample (d) Inference time per sample

Figure 7: CDF of the accuracy produced by IGNNITION and the original TensorFlow implementations (a and b); Comparison of the execution cost of both types of implementations, during training and inference (c and d).

mentations of IGNNITION are equivalently accurate and efficient to native TensorFlow implementations. This reveals the capability of this framework to produce efficient GNN models, while offering substantial time savings for non-ML experts to implement them. They can just achieve it by filling the YAML model description file described in Sec. 3.1. Moreover, users can leverage additional functionalities such as the visual representations of the debugging system (Fig. 6) and the advanced error-checking mechanisms to easily fix potential bugs.

5 Related Work

Apart from IGNNITION, some other programming libraries support the implementation of GNNs. Particularly, two main types of libraries can be differentiated. On the one hand, libraries like GraphGym (You et al., 2020) or Spektral (Grattarola and Alippi, 2020) aim to simplify the GNN design process. In the case of GraphGym, for instance, through a codeless interface. This approach, however, comes at the expense of strict assumptions on GNN architectures, limiting significantly the variety of GNNs that can be supported. For instance, these solutions do not offer easy support to implement state-of-the-art GNN models applied to computer networks, such as (Rusek et al., 2019) or (Badia-Sampera et al., 2019), given their non-standard message-passing architectures (e.g., various entity types and multi-stage message passing). On the other hand, libraries such as Graph Nets (Battaglia et al., 2018), PyTorch Geometric (Fey and Lenssen, 2019), or DGL (Wang et al., 2019) do not impose such restrictions. To achieve this, they strongly rely on generic ML libraries (e.g., TensorFlow, PyTorch) to implement some critical parts of the model, which considerably hinders the implementation of GNN models for users with little experience in neural network programming.

Table 1: Comparison of existing programming libraries with support for GNN.

Name	High-level GNN abstraction	Support for non-standard GNNs	Debugging assistant
IGNNITION	✔	✔	✔
Graph Nets (Battaglia et al., 2018)	✗	✔	✗
PyTorch Geometric (Fey and Lenssen, 2019)	✗	✔	✗
DGL (Wang et al., 2019)	✗	✔	✗
Spektral (Grattarola and Alippi, 2020)	✗	✗	✗
GraphGym (You et al., 2020)	✔	✗	✗

Table 1 summarizes the main differences between IGNNITION and the main existing libraries with support for GNN. With IGNNITION, users can define GNNs using a *high-level abstraction* that avoids mathematical formulation and coding tensor-based operations. More importantly, this framework puts the spotlight on complex GNN models with non-standard message-passing architectures, while avoiding any performance penalty or requiring users to write a single line of TensorFlow code. To the best of our knowledge, this is not possible with any other existing library.

6 Conclusion

In this paper, we introduced IGNNITION, an open-source framework based on TensorFlow that enables fast prototyping of *Graph Neural Networks*. IGNNITION works over a novel high-level abstraction called the *Multi-Stage Message Passing graph* (MSMP graph), which isolates users from the complex mathematical formulations and tensor-based operations used in traditional ML libraries (e.g., TensorFlow). MSMP graphs are flexible enough to support any state-of-the-art GNN architecture with complex message passing strategies. We implemented two state-of-the-art GNN models applied to computer networks with IGNNITION to show its versatility. Likewise, we validated that the performance of the GNNs produced by this framework is equivalent to that of native TensorFlow implementations.

Acknowledgement

This work has received funding from the European Union's Horizon 2020 research and innovation programme within the framework of the NGI-POINTER Project funded under grant agreement No. 871528. This paper reflects only the authors' view; the European Commission is not responsible for any use that may be made of the information it contains. This work was also supported by the Spanish MINECO under contract TEC2017-90034-C2-1-R (ALLIANCE) and the Catalan Institution for Research and Advanced Studies (ICREA).

REFERENCES

Abadi, M., et al., 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 .

Badia-Sampera, A., et al., 2019. Towards more realistic network models based on graph neural networks, in: Proceedings of the ACM CoNEXT student workshop, pp. 14–16.

Battaglia, P., et al., 2016. Interaction networks for learning about objects, relations and physics, in: Advances in neural information processing systems (NIPS), pp. 4502–4510.

Battaglia, P., et al., 2018. Relational inductive biases, deep learning, and graph networks. arXiv preprint arXiv:1806.01261 .

BNN Center, 2021. IGNNITION - Documentation. URL: https://ignnition.net/doc/.

Chollet, F., et al., 2015. Keras. URL: https://github.com/fchollet/keras.

Farrell, S., et al., 2018. Novel deep learning methods for track reconstruction. arXiv preprint arXiv:1810.06111 .

Feng, Y., You, H., Zhang, Z., Ji, R., Gao, Y., 2019. Hypergraph neural networks, in: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 3558–3565.

Fey, M., Lenssen, J.E., 2019. Fast graph representation learning with pytorch geometric. arXiv preprint arXiv:1903.02428 .

Gainza, P., et al., 2020. Deciphering interaction fingerprints from protein molecular surfaces using geometric deep learning. Nature Methods 17, 184–192.

Geyer, F., Carle, G., 2018. Learning and generating distributed routing protocols using graph-based deep learning, in: Proceedings of the ACM SIGCOMM BigDAMA Workshop, pp. 40–45.

Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E., 2017. Neural message passing for quantum chemistry, in: Proceedings of the International Conference on Machine Learning (ICML), pp. 1263–1272.

Grattarola, D., Alippi, C., 2020. Graph neural networks in tensorflow and keras with spektral. arXiv preprint arXiv:2006.12138 .

Hagberg, A.A., Schult, D.A., Swart, P.J., 2008. Exploring network structure, dynamics, and function using networkx, in: Python in Science Conference, pp. 11 – 15.

Kipf, T.N., Welling, M., 2017. Semi-supervised classification with graph convolutional networks, in: Proceedings of the International Conference on Learning Representations (ICLR).

Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S., 2016. Gated graph sequence neural networks, in: Proceedings of the International Conference on Learning Representations (ICLR).

Liang, X., Shen, X., Feng, J., Lin, L., Yan, S., 2016. Semantic object parsing with graph LSTM, in: European Conference on Computer Vision, Springer. pp. 125–143.

Paszke, A., et al., 2019. Pytorch: An imperative style, high-performance deep learning library, in: Advances in neural information processing systems (NIPS), pp. 8026–8037.

Prates, M.O., Avelar, P.H., Lemos, H., Gori, M., Lamb, L., 2019. Typed graph networks. arXiv preprint arXiv:1901.07984 .

Rusek, K., et al., 2019. Unveiling the potential of graph neural networks for network modeling and optimization in sdn, in: Proceedings of the ACM Symposium on SDN Research (SOSR), pp. 140–151.

Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G., 2008. The graph neural network model. IEEE Transactions on Neural Networks 20, 61–80.

Veličković, P., et al., 2017. Graph attention networks. arXiv preprint arXiv:1710.10903 .

Wang, M., et al., 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315 .

Ying, R., et al., 2018. Graph convolutional neural networks for web-scale recommender systems, in: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 974–983.

You, J., Liu, B., Ying, Z., Pande, V., Leskovec, J., 2018. Graph convolutional policy network for goal-directed molecular graph generation, in: Advances in Neural Information Processing Systems (NIPS), pp. 6410–6421.

You, J., Ying, R., Leskovec, J., 2020. Design space for graph neural networks, in: NeurIPS.

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., Sun, M., 2018. Graph neural networks: A review of methods and applications. arXiv preprint arXiv:1812.08434 .

Zitnik, M., Agrawal, M., Leskovec, J., 2018. Modeling polypharmacy side effects with graph convolutional networks. Bioinformatics 34, i457–i466.