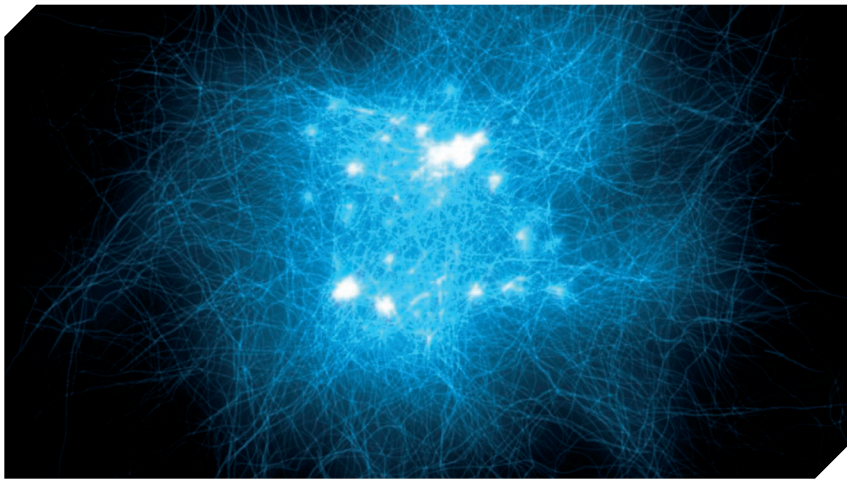
Daniele Grattarola
Università della Svizzera italiana, SWITZERLAND

Cesare Alippi
Politecnico di Milano, ITALY and Università della Svizzera italiana, SWITZERLAND

Graph Neural Networks in TensorFlow and Keras with Spektral

Abstract

Graph neural networks have enabled the application of deep learning to problems that can be described by graphs, which are found throughout the different fields of science, from physics to biology, natural language processing, telecommunications or medicine. In this paper we present Spektral, an open-source Python library for building graph neural networks with TensorFlow and the Keras application programming interface. Spektral implements a large set of methods for deep learning on graphs, including message-passing and pooling operators, as well as utilities for processing graphs and loading popular benchmark datasets. The purpose of this library is to provide the essential building blocks for creating graph neural networks, focusing on the guiding principles of user-friendliness and quick prototyping on which Keras is based. Spektral is, therefore, suitable for absolute beginners and expert deep learning practitioners alike. In this work, we present an overview of Spektral's features and report the performance of the methods implemented by the library in scenarios of node classification, graph classification, and graph regression.



©SHUTTERSTOCK.COM/VOLODIMIR ZOZULINSKYI

I. Introduction

Graph Neural Networks (GNNs) are a class of deep learning methods designed to perform inference on data described by graphs [1]. When learning on graphs, the relations among the entities that constitute the input domain provide a useful inductive bias that can be leveraged in the prediction. For instance, one could be interested in classifying the users of a social network according to their friendships and interactions. In this setting, the atomic entities of the learning problem are the individual nodes, and the relations existing among nodes provide valuable information for the *node-level* inference (Fig. 1(a)). On the other hand, a different task could be to predict the chemical properties of molecules from their molecular graph. In this case, the relations (chemical bonds) between the atoms define global properties of the molecules, and the inference task is at the *graph level* (Fig. 1(b)).

Due to the different possibilities offered by graph machine learning and the large number of applications where graphs are naturally found, GNNs have been successfully applied to a diverse spectrum of fields to solve a variety of

Digital Object Identifier 10.1109/MCI.2020.3039072
Date of current version: 12 January 2021

Corresponding Author: Daniele Grattarola (e-mail: daniele.grattarola@usi.ch).

tasks. In physics, GNNs have been used to learn physical models of complex systems of interacting particles [2]–[5]. In recommender systems, the interactions between users and items can be represented as a bipartite graph and the goal is to predict new potential edges (i.e., which items could a user be interested in), which can be achieved with GNNs [6], [7]. GNNs have also been largely applied to the biological sciences, with applications ranging from the recommendation of medications [8], to the prediction of protein-protein and protein-ligand interactions [9], and in chemistry, for the prediction of quantum molecular properties as well as the generation of novel compounds and drugs [10], [11]. Finally, GNNs have been successfully applied in fields like natural language processing [12], [13] and even more complex tasks like abstract reasoning [14]–[16] and decision making with reinforcement learning [17], [18].

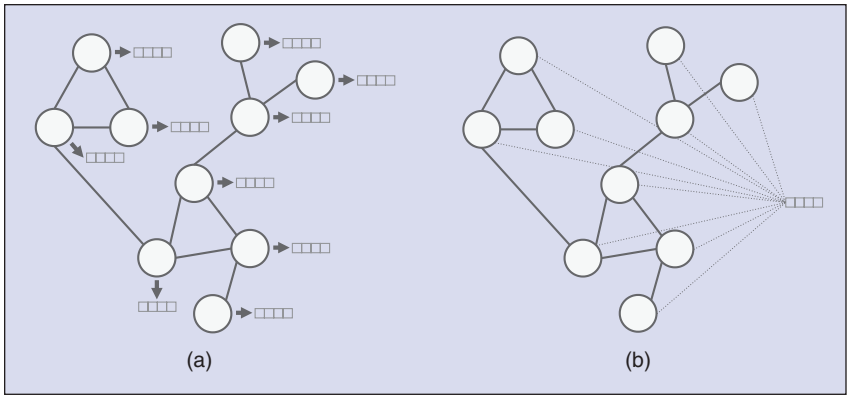


FIGURE 1 Schematic representation of (a) node-level prediction and (b) graph-level prediction. When predicting node properties, the GNN predicts an output for each individual node (or a subset thereof). When predicting graph-level properties, the GNN outputs a single prediction for the entire graph.

At the core of GNNs there are two main types of operations, which can be interpreted as a generalization of the convolution and pooling operators in convolutional neural networks: *message passing* and *graph pooling* (Fig. 2). The former is used to learn a non-linear transformation of the input graphs and the latter to reduce their size. When combined, these two operations enable graph representation learning as a general tool to predict node-level, edge-level, and global properties of graphs. Several works in recent literature have introduced models for either message passing [19]–[28] or graph pooling [29]–[32].

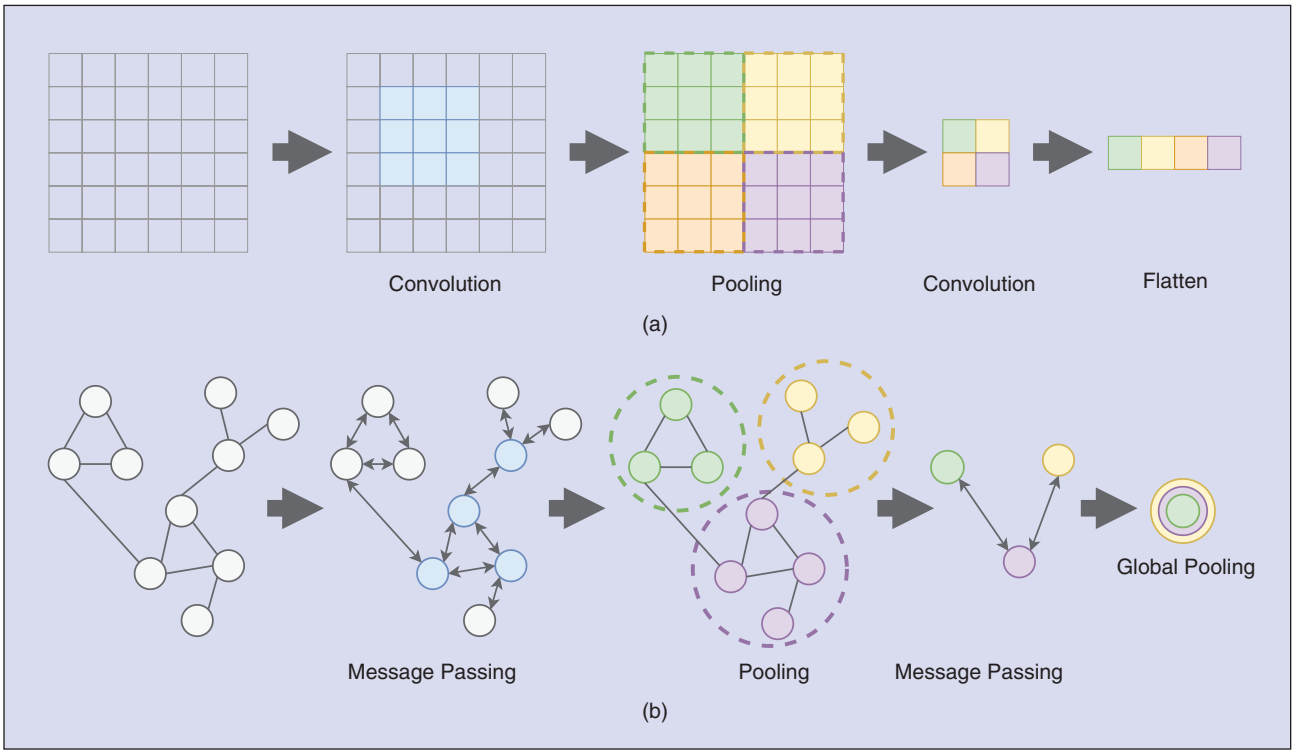


FIGURE 2 Comparison of a (a) convolutional neural network and a (b) graph neural network with message-passing, pooling, and global pooling layers. The role of message-passing layers is to compute a representation of each node in the graph, leveraging *local* information (*messages*) from its neighbours, similarly to how traditional convolutional layers use local receptive fields to compute a representation for each pixel of an image (the receptive fields for a single node and a single pixel are highlighted in blue). The role of pooling layers is to reduce the size of the graph by aggregating or discarding redundant information, so that the GNN can learn a hierarchical representation of the input data. Finally, global pooling layers reduce the graph to a single vector, usually to feed it as input to a multi-layer perceptron for classification or regression, which is an equivalent operation to flattening the feature maps in convolutional networks.

Thanks to the increasing popularity of GNNs, many software libraries implementing the building blocks of graph representation learning have been developed in recent years, paving the way for the adoption of GNNs in other fields of science. One of the major challenges faced by researchers and software developers who wish to contribute to the larger scientific community is to make software both accessible and intuitive, so that even non-technical audiences can benefit from the advances carried by intelligent systems. In this spirit, Keras is an application programming interface (API) for creating neural networks, developed according to the guiding principle that "being able to go from idea to result with the least possible delay is key to doing good research" [33]. Keras is designed to reduce the cognitive load of end users, shifting the focus away from the boilerplate implementation details and allowing instead to focus on the creation of models. As such, Keras is extremely beginner-friendly and, for many, an entry point to machine learning itself. At the same time, Keras integrates smoothly with its TensorFlow [34] backend and enables users to build any model that they could have implemented in pure TensorFlow. This flexibility makes Keras an excellent tool even for expert deep learning practitioners and has recently led to TensorFlow's adoption of Keras as the official interface to the framework.

In this paper we present Spektral, a Python library for building graph neural networks using TensorFlow and the Keras API. Spektral implements some of the most important papers from the GNN literature as Keras layers, and it integrates seamlessly within Keras models and with the most important features of Keras like the training loop, callbacks, distributed training, and automatic support for GPUs and TPUs. As such, Spektral inherits the philosophy of ease of use and flexibility that characterizes Keras. The components of Spektral act as standard TensorFlow operations and can be easily used even in more advanced settings, integrating tightly with all the features of TensorFlow and allowing for easy deployment to production systems. For these reasons, Spektral is the ideal library to implement GNNs in the TensorFlow ecosystem, both for total beginners and experts alike.

All features of Spektral are documented in detail[1] and a collection of examples is provided with the source code. The project is released on GitHub[2] under MIT license.

II. Library Overview

A. Representing Graphs

Let $\mathcal{G} = \{\mathcal{X}, \mathcal{E}\}$ be a graph where $\mathcal{X} = \{\mathbf{x}_i \in \mathbb{R}^F | i = 1, \dots, N\}$ is the set of nodes with F-dimensional real attributes, and $\mathcal{E} = \{\mathbf{e}_{ij} \in \mathbb{R}^S | \mathbf{x}_i, \mathbf{x}_j \in \mathcal{X}\}$ the set of edges with S dimensional real attributes. In Spektral, we represent $\mathcal{G}$ by its binary adjacency matrix $\mathbf{A} \in \{0, 1\}^{N \times N}$, node features $\mathbf{X} \in \mathbb{R}^{N \times F}$, and edge features $\mathbf{E} \in \mathbb{R}^{N \times N \times S}$. Any format accepted by Keras to represent the above matrices is also supported by Spektral, which means that it also natively supports the NumPy stack of scientific computing libraries for Python. Most of the layers and utilities implemented in Spektral also support the sparse matrices of the SciPy library, making them computationally efficient both in time and memory.

B. Data Modes

Spektral makes very few assumptions on how a user may want to represent graphs and transparently deals with batches of graphs represented as higher-order tensors or disjoint unions. The library supports four different ways of representing graphs (or batches thereof), which we refer to as *data modes*.

In *single mode*, the data describes a single graph with its adjacency matrix and attributes, and inference usually happens at the level of individual nodes. *Disjoint mode* is a special case of single mode, where the graph is obtained as the disjoint union of a set of smaller graphs. In this case the node attributes of the graphs are stacked in a single matrix and their adjacency matrices are combined in a block-diagonal matrix. This is a practical way of representing batches of variable-order graphs, although it requires an additional data structure to keep track of the different components. Alternatively, in *batch mode*, a set of graphs is represented by stacking their adjacency and attributes matrices in higher-order tensors of shape $B \times N \times \dots$. This mode is akin to traditional batch processing in machine learning and can be more naturally adopted in deep learning architectures. However, it requires the graphs in a batch to have the same number of nodes. Finally, *mixed mode* is the one most often found in traditional machine learning literature and consists of a graph with fixed support but variable attributes. Common examples of this mode are found in computer vision, where images have a fixed 2-dimensional grid support and only differ in the pixel values (i.e., the node attributes), and in traditional graph signal processing applications.

The `utils` module exposes some useful utilities for manipulating the data and converting between the different data modes.

C. Message Passing

Message-passing networks are a general paradigm introduced by Gilmer et al. [19] that unifies most GNN methods found in the literature as a combination of *message*, *aggregation*, and *update* functions. Message-passing layers are equivalent in role to the convolutional operators in convolutional neural networks, and are the essential components of graph representation learning. Message-passing layers in Spektral are available in the

FIGURE 3 The stylised ghost logo of Spektral.

[1] https://graphneural.network
[2] https://github.com/danielegrattarola/spektral

`layers.convolutional` module.[3] Currently, Spektral implements fifteen different message-passing layers including Graph Convolutional Networks (GCN) [22], ChebNet [21], GraphSAGE [25], ARMA convolutions [26], Edge-Conditioned Convolutions (ECC) [23], Graph Attention Networks (GAT) [24], APPNP [28], and Graph Isomorphism Networks (GIN) [27], as well as the methods proposed by Li et al. [35], 36], Thekumparampil et al. [37], Du et al. [38], Xie and Grossman [39], Wang et al. [40] and a general interface that can be extended to implement message-passing layers. The available methods are sufficient to deal with all kinds of graphs, including those with attributed edges.

D. Graph Pooling

Graph pooling refers to any operation to reduce the number of nodes in a graph and has a similar role to pooling in traditional convolutional networks for learning hierarchical representations. Since pooling computes a coarser version of the graph at each step, ultimately resulting in a single vector representation, it is usually applied to problems of graph-level inference. Graph pooling layers are available in `layers.pooling` and include: DiffPool [29], MinCut pooling [31], Top-K pooling [30], 32], and Self-Attention Graph Pooling (SAGPool) [41]. Spektral also implements *global* graph pooling methods, which can be seen as a limit case of graph pooling where a graph is reduced to a single node, i.e., its node features are reduced to a single vector. Spektral implements six different global pooling strategies: sum, average, max, gated attention (GAP) [36], SortPool [42], and attention-weighted sum (AWSP).[4]

E. Datasets

Spektral comes with a large variety of popular graph datasets accessible from the `datasets` module. The datasets available from Spektral provide benchmarks for transductive and inductive node classification, graph signal classification, graph classification, and graph regression. In particular, the following datasets can be loaded with Spektral: the citation networks, Cora, CiteSeer, and Pubmed [43]; the protein-protein interaction dataset (PPI) [25], [44], [45] and the Reddit communities network dataset [25] from the GraphSAGE paper [25]; the QM9 chemical dataset of small molecules [46]; the MNIST 8-NN graph for graph signal classification as proposed by Defferrard et al. [21]; the Benchmark Data Sets for Graph Kernels [47]. Each dataset is automatically downloaded and stored locally when necessary.

F. Implementing Message-Passing Layers

Spektral offers a large set of utilities that can be used by advanced users to create new GNN layers. The `layers.ops` module implements wrappers for common matrix operations that automatically handle sparse inputs, data modes, and batches of graphs, as well as functions to compute the characteristic graph matrices in TensorFlow.

While creating layers that support all four data modes requires the users to define the case-specific behaviour, Spektral also implements a general `MessagePassing` class to quickly implement message-passing networks as presented by Gilmer et al. [19], in single and disjoint mode with sparse inputs. These have the form

$$\mathbf{z}_i = \gamma\big(\mathbf{x}_i, \square_{j\in\mathcal{N}(i)}\,\phi(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{ji})\big),$$

where γ is a differentiable update function, ϕ is a differentiable message function, and $\square$ is a permutation-invariant function (like the sum or the average) to aggregate the messages over the neighborhood $\mathcal{N}(i)$ of node i. By overriding the behaviour of the corresponding class methods, users can easily define new GNN layers and use them for both node-level and graph-level inference tasks.

G. Technical Notes

Spektral is distributed through the Python Package Index (package name: `spektral`), supports all UNIX-like platforms,[5] and has no proprietary dependencies. The library is compatible with Python version 3.5 and above. Starting from version 0.2, Spektral is developed for TensorFlow 2 and its integrated implementation of Keras. Version 0.1 of Spektral, which is based on TensorFlow 1 and the standalone version of Keras, will be maintained until TensorFlow 1 is officially discontinued, although new features will only be added to the newer versions of Spektral.

III. Comparison to Other Libraries

Given the growing popularity of the field, several libraries for GNNs have appeared in recent years. Among the most notable, we cite PyTorch Geometric[6] (PyG) [48] and the Deep Graph Library[7] (DGL) [49], both of which are based on the PyTorch deep learning library.[8] Instead, Spektral is specifically developed for the TensorFlow ecosystem, which to this day is estimated to support the majority of deep learning applications both in research and industry [50]. The features offered by Spektral, summarised in Table I, are largely similar to those offered both by PyG (which however implements a much larger variety of message-passing methods and other algorithms from GNN literature) and by DGL. The `MessagePassing` interface of Spektral is based on a gather-scatter paradigm similar to that of PyG, using native TensorFlow operations to quickly access a node's neighborhood and aggregating messages. The computational performance of Spektral's layers is therefore comparable to that of PyG, with small differences due to implementation details and differences between the two respective backend frameworks. We also mention

[3]The name *convolutional* derives from the homonymous module in Keras, as well as message-passing layers being originally derived as a generalisation of convolutional operators.

[4]While never published in the literature, attention-weighted sum is a straightforward concept that consists of computing a weighted sum of the node features, where the weights are computed through a simple attentional mechanism

[5]It is also largely compatible with Windows.

[6]https://pytorch-geometric.readthedocs.io/

[7]https://docs.dgl.ai/

[8]Note that DGL also supports MXNet and TensorFlow as backends, albeit with a reduced set of features. In this paper, we consider DGL to be a PyTorch library.

the StellarGraph library for GNNs which, like Spektral, is based on Keras. This library implements six message-passing layers, four of which are available in Spektral (GCN, GraphSAGE, GAT and APPNP), but does not offer pooling layers and relies on a custom format for graph data, which limits flexibility. Finally, the `graph_nets` package[9] implements Graph Networks as proposed by Battaglia et al. [1]. However, the package is not a full library for GNNs and only offers a general interface that users can extend.

Spektral is currently the most feature-rich library for GNNs in TensorFlow. The extensive collection of examples and tutorials, the Keras integration, and the transparent handling of different data modes make it extremely easy for new users to familiarize with GNNs, while its good computational performance and variety of available methods make Spektral a sensible choice even for advanced use cases.

IV. Applications

In this section, we report some experimental results on several well-known benchmark tasks, in order to provide a high-level overview of how the different methods implemented by Spektral perform in a standard research use case scenario. We report the results for three main settings: a node classification task and two tasks of graph-level property prediction, one of classification and one of regression. All experimental details are reported in Appendix A.

A. Node Classification

In our first experiment, we consider a task of semi-supervised node classification on the Cora, CiteSeer, and Pubmed citation networks. In these datasets, nodes represent text documents and the undirected edges represent citations. The task consists of classifying the documents into a finite number of subject areas. We evaluate GCN [22], ChebNet [21], ARMA [26], GAT [24] and APPNP [28]. We reproduce the same experimental settings described in the original papers, but we use the random data splits suggested by Shchur et al. [51] for a fairer evaluation.

B. Graph Classification

To evaluate the pooling layers, DiffPool [29], MinCut [31], Top-*K* [30], and SAGPool [41], we consider a task of graph-level classification, where each graph represents an individual sample to be classified. We use four datasets from the Benchmark Data Sets for Graph Kernels: Proteins, IMDB-Binary, Mutag and NCI1. Here, we adopt a fixed GNN architecture (described in Appendix A) where we only change the pooling method. To assess whether each pooling layer is actually beneficial for learning a representation of the data, we also evaluate the same GNN without pooling (*Flat*).

C. Graph Regression

To evaluate the global pooling methods, we consider the task of predicting quantum molecular properties using the QM9 database of small molecules. We train a GNN on four different continuous targets for graph-level regression: dipole moment (Mu), isotropic polarizability (Alpha), energy of HOMO (Homo), and internal energy at OK (U0).

Since the molecules in QM9 have attributed edges, we adopt a GNN based on ECC, which is designed to integrate edge attributes in the message-passing operation. We note that the architecture used for this experiment is significantly smaller and simpler than the current state of the art [52], and that these results are only meant to show a comparison between the different global pooling methods, rather than replicating the exact performance figures of other works (which may be significantly different than what we report).

D. Results

The results for each experiment are reported in Tables II, III, and IV. In the

TABLE I Comparison of different GNN libraries. The *Framework* column indicates the backend framework supported by the library, while the *MP* and *Pooling* columns indicate the number of different message-passing and pooling layers implemented by the library, respectively.

LIBRARY	FRAMEWORK	MP	POOLING
Spektral	TensorFlow	15	10
PyG	PyTorch	28	14
DGL	PyTorch, others	15	7
`graph_nets`	TensorFlow	1	N/A
StellarGraph	TensorFlow	6	N/A

TABLE II Classification accuracy on the node classification tasks.

Dataset	ChebNet	GCN	GAT	ARMA	APPNP
CORA	77.4 ± 1.5	79.4 ± 1.3	82.0 ± 1.2	80.5 ± 1.2	**82.8** ± 0.9
CITESEER	68.2 ± 1.6	68.8 ± 1.4	70.0 ± 1.0	**70.6** ± 0.9	70.0 ± 1.0
PUBMED	74.0 ± 2.7	76.6 ± 2.5	73.8 ± 3.3	77.2 ± 1.6	**78.2** ± 2.1

TABLE III Classification accuracy on the graph classification tasks.

Dataset	Flat	MinCut	DiffPool	Top-*K*	SAGPool
Proteins	74.3 ± 4.5	**75.5** ± 2.0	74.1 ± 3.9	70.5 ± 3.4	71.3 ± 5.0
IMDB-B	72.8 ± 7.2	**73.6** ± 5.4	70.6 ± 6.6	67.7 ± 8.2	69.3 ± 5.7
Mutag	72.5 ± 14.0	81.4 ± 10.7	**83.5** ± 9.7	79.2 ± 8.0	78.5 ± 8.3
NCI1	**77.3** ± 2.6	74.4 ± 1.9	71.1 ± 3.0	72.0 ± 3.0	69.4 ± 8.4

[9]https://github.com/deepmind/graph_nets

first experiment, results are compatible with what reported in the literature, although some differences in performance are present due to the use of a random data split rather than the predefined one used in the original experiments. The APPNP operator consistently achieves good results on the citation networks, outperforming the other methods on Cora and Pubmed, and coming close to ARMA on CiteSeer. For graph classification, the results are sometimes different than what is reported in the literature, due to the standardised architecture that we used in this experiment. MinCut generally achieves the best performance followed by DiffPool. We also note that the Flat baseline often achieves better results than the

TABLE IV Mean-squared error on the graph regression tasks. Results for Homo are in scale of 10^{-5}.

DATASET	AVERAGE	SUM	MAX	GAP	AWSP
MU	1.12 ± 0.03	1.02 ± 0.02	**0.90** ± 0.04	1.04 ± 0.05	0.99 ± 0.03
ALPHA	3.15 ± 0.65	2.38 ± 0.64	6.20 ± 0.33	**1.89** ± 0.59	31.1 ± 0.37
HOMO	9.24 ± 0.41	9.22 ± 0.51	8.90 ± 0.36	9.04 ± 0.29	**8.05** ± 0.29
U0	0.42 ± 0.14	0.50 ± 0.13	110.7 ± 4.5	**0.22** ± 0.13	624.0 ± 19.0

Appendix

Experimental Details

This section summarises the architectures and hyperparameters used in the experiments of Section IV.

A. Node Classification

Hyperparameters:

- ❏ Learning rate: see original papers;
- ❏ Weight decay: see original papers;
- ❏ Epochs: see original papers;
- ❏ Patience: see original papers;
- ❏ Repetitions per method and per dataset: 100;
- ❏ Data: we used Cora, Citeseer and Pubmed. As suggested in [51], we use random splits with 20 labels per class for training, 30 labels per class for early stopping, all the remaining labels for testing.

B. Graph Classification

We configure a GNN with the following structure: GCS - Pooling - GCS - Pooling - GCS - GlobalSumPooling - Dense, where GCS indicates a *Graph Convolutional Skip* layer as described in [26], Pooling indicates the graph pooling layer being tested, GlobalSumPooling represents a global sum pooling layer, and Dense represents the fully-connected output layer. GCS layers have 32 units each, ReLU activation, and L2 regularisation applied to both weight matrices. The same L2 regularisation is applied to pooling layers when possible. Top-K and SAGPool layers are configured to output half of the nodes for each input graph. DiffPool and MinCut are configured to output $K = \bar{N}/2$ nodes at the first layer, and $K = \bar{N}/4$ nodes at the second layer, where $\bar{N}$ is the average order of the graphs in the dataset. When using DiffPool, we remove the first two GCS layers, because DiffPool has an internal message-passing layer for the input features. DiffPool and MinCut were trained in batch mode by zero-padding the adjacency and node attributes matrices. All networks were trained using Adam with the default parameters of Keras, except for the learning rate.

Hyperparameters:

- ❏ Batch size: 8;
- ❏ Learning rate: 0.001;
- ❏ Weight decay: 0.00001;
- ❏ Epochs: models trained to convergence;
- ❏ Patience: 50 epochs;
- ❏ Repetitions per method and per dataset: 10;
- ❏ Data: we used the Benchmark Datasets for Graph Kernels as described in [53], that were modified to contain no isomorphic graphs. For each run, we randomly split the dataset and use 80% of the data for training, 10% for early stopping, and 10% for testing.

C. Graph Regression

We configure a GNN with the following structure: ECC - ECC - GlobalPooling - Dense, where ECC indicates an Edge-Conditioned Convolutional layer [23] and GlobalPooling indicates the global pooling layer being tested. ECC layers have 32 units each, and ReLU activation. No regularisation is applied to the GNN. GAP is configured to use 32 units. All networks were trained using Adam with the default parameters of Keras, except for the learning rate. We use the mean squared error as loss.

Node features are one-hot encodings of the atomic number of each atom. Edge features are one-hot encodings of the bond type. The units of measurement for the target variables are: debye units (D) for Mu, a_0^3 (a_0 is the Bohr radius) for Alpha, and Hartree (Ha) for Homo and U0 [46].

Hyperparameters:

- ❏ Batch size: 32;
- ❏ Learning rate: 0.0005;
- ❏ Epochs: models trained to convergence;
- ❏ Patience: 10 epochs;
- ❏ Repetitions per method and per dataset: 5;
- ❏ Data: for each run, we randomly split the dataset and use 80% of the molecules for training, 10% for early stopping, and 10% for testing.

equivalent GNNs equipped with pooling. For graph regression, results show that the choice of global pooling method can have a significant impact on performance. In particular, the GAP operator performs best on Alpha and U0, while the best results on Mu and Homo are obtained with max pooling and AWSP, respectively. However, we note how these latter two operators have largely unstable performances depending on the datasets, as both fail on Alpha and U0.

E. Execution Times

In an empirical test similar to that conducted by Fey and Lenssen [48], we measured the execution time of Spektral and PyG (which is the most efficient among the PyTorch libraries [48]) for training GCN, ChebNet and GAT on the citation networks. Each test consisted of running 200 training epochs with the models and hyperparameters described by Kipf and Welling [22] and Velickovic et al. [24]. The tests were repeated 10 times per dataset, on an Nvidia GeForce GTX 1050 with 4GB of video memory. The execution times are reported in Table V. The results indicate a comparable computational performance between the two frameworks, with Spektral's implementation of GCN and ChebNet being faster than PyG's, and *vice versa* for GAT. Both libraries, however, are well within the same order of magnitude indicating that the differences are minor and likely due to the backend frameworks.

TABLE V Comparison of the execution times of Spektral and PyG for 200 training epochs with GCN, ChebNet, and GAT on the citation networks. The *Change* column indicates the difference between the two libraries (a negative change means that Spektral was faster). Best execution times are in bold.

MODEL	DATASET	PYG	SPEKTRAL	CHANGE
GCN	CORA	0.332s ± 0.002	**0.183s** ± 0.002	**– 44.9%**
	CITESEER	0.488s ± 0.009	**0.396s** ± 0.011	**– 18.8%**
	PUBMED	0.834s ± 0.004	**0.683s** ± 0.001	**– 18.1%**
CHEBNET	CORA	4.690s ± 0.007	**2.059s** ± 0.008	**– 56.0%**
	CITESEER	11.441s ± 0.165	**5.470s** ± 0.006	**– 52.2%**
	PUBMED	12.517s ± 0.148	**6.221s** ± 0.004	**– 50.2%**
GAT	CORA	**1.527s** ± 0.002	2.042s ± 0.074	+ 33.7%
	CITESEER	**2.032s** ± 0.003	3.427s ± 0.085	+ 68.6%
	PUBMED	**7.427s** ± 0.014	10.63s ± 0.132	+ 43.1%

F. Code Example

Since Spektral is built as a Keras extension, its main building blocks are designed

Box 1: Implementing a Model for Graph Classification

The following code snippet shows how to define a simple graph classifier in batch mode, using Spektral and the Keras API. Note how Spektral's layers can be combined with the standard layers of Keras using the functional API. In particular, this model is composed of a GCN layer [22], a MinCutPool layer [31], a global sum pooling layer, and finally a fully-connected classifier:

```
n = ...    # number of nodes
f = ...    # size of input attributes

x_in = Input(shape=(n, f))                          # input attributes
a_in = Input(shape=(n, n))                          # input adjacency matrix
x_1 = GraphConv(32, activation='relu')([x_in, a_in])    # update node attributes
x_2, a_2 = MinCutPool(k=n//2)([x_1, a_in])          # pool nodes and edges
pool = GlobalSumPool()(x_2)                         # global pooling
output = Dense(n_out, activation='softmax')(pool)       # fully-connected classifier

model = Model([x_in, a_in], output)                 # create model
model.compile('adam', 'categorical_crossentropy')       # use Adam and cross-entropy
```

To train this model, assuming to have the training graphs and labels stored as Numpy arrays as described in Section II-A, the built-in training loop of Keras can be used:

```
model.fit([x_train, a_train], y_train)                  # train the model
```

Similarly, evaluation and inference can be done with:

```
score = model.evaluate([x_test, a_test], y_test)        # evaluate on test set
prediction = model.predict([x_new, a_new])              # predict on new samples
```

to integrate seamlessly in Keras models using the same intuitive and familiar API. For example, adding a GCN layer with 16 units and ReLU activation to a Keras model using the functional API is as simple as calling:

```
x_out = GraphConv(16, 'relu')\
        ([x_in, a_in])
```

where the layer takes as input the node attributes (`x_in`) and the normalised adjacency matrix (`a_in`) [22].

Creating more complex architectures is then just a matter of combining the existing building blocks to obtain a Keras model. For instance, Box 1 shows how to create a model similar to the one used in Section IV-B, with message-passing, pooling and global pooling layers.

V. Conclusion

We presented Spektral, a library for building graph neural networks using the Keras API. Spektral implements several state-of-the-art methods for GNNs, including message-passing and pooling layers, a wide set of utilities, and comes with many popular graph datasets. The library is designed for providing a streamlined user experience and is currently the most mature library for GNNs in the TensorFlow ecosystem. In the future, we will keep Spektral up to date with the ever-growing field of GNN research, and we will focus on improving the performance of its core components.

References

[1] P. W. Battaglia et al., "Relational inductive biases, deep learning, and graph networks," 2018, arXiv:1806.01261.

[2] P. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, and K. Kavukcuoglu "Interaction networks for learning about objects, relations and physics," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 4502–4510.

[3] T. Kipf, E. Fetaya, K.-C. Wang, M. Welling, and R. Zemel, "Neural relational inference for interacting systems," 2018, arXiv:1802.04687.

[4] A. Sanchez-Gonzalez et al., "Graph networks as learnable physics engines for inference and control," 2018, arXiv:1806.01242.

[5] S. Farrell et al., "Novel deep learning methods for track reconstruction," 2018.

[6] R. v. d. Berg, T. N. Kipf, and M. Welling, "Graph convolutional matrix completion," 2017, arXiv:1706.02263.

[7] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 974–983.

[8] J. Shang, C. Xiao, T. Ma, H. Li, and J. Sun, "Gamenet: graph augmented memory networks for recommending medication combination," in *Proc. AAAI Conf. Artific. Intell.*, vol. 33, 2019, pp. 1126–1133. doi: 10.1609/aaai.v33i01.33011126.

[9] P. Gainza et al., "Deciphering interaction fingerprints from protein molecular surfaces using geometric deep learning," *Nature Methods*, vol. 17, no. 2, pp. 184–192, 2020. doi: 10.1038/s41592-019-0666-6.

[10] K. Do, T. Tran, and S. Venkatesh, "Graph transformation policy network for chemical reaction prediction," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 750–760. doi: 10.1145/3292500.3330958.

[11] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec, "Graphrnn: Generating realistic graphs with deep auto-regressive models," 2018, arXiv:1802.08773.

[12] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," 2018, arXiv:1811.01824.

[13] N. De Cao, W. Aziz, and I. Titov, "Question answering by reasoning across documents with graph convolutional networks," 2018, arXiv:1808.09920.

[14] A. Santoro, D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap, "A simple neural network module for relational reasoning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 4967–4976.

[15] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," 2017, arXiv:1711.00740.

[16] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *Proc. Euro Semantic Web Conf.*, Springer, 2018, pp. 593–607.

[17] V. Zambaldi et al., "Relational deep reinforcement learning," 2018, arXiv:1806.01830, 2018.

[18] J. B. Hamrick et al., "Relational inductive bias for physical construction in humans and machines," 2018, arXiv:1806.01203.

[19] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," 2017, arXiv:1704.01212.

[20] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, 2009. doi: 10.1109/TNN.2008.2005605.

[21] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 3844–3852.

[22] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," presented at the Int. Conf. Learn. Represent. (ICLR), 2016.

[23] M. Simonovsky and N. Komodakis, "Dynamic edge-conditioned filters in convolutional neural networks on graphs," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit*, 2017.

[24] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," 2017, arXiv:1710.10903.

[25] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 1024–1034.

[26] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi, "Graph neural networks with convolutional ARMA filters," 2019, arXiv:1901.01343.

[27] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" presented at the Int. Conf. Learn Represent. (ICLR), 2019.

[28] J. Klicpera, A. Bojchevski, and S. Günnemann, "Predict then propagate: Graph neural networks meet personalized pagerank," presented at the Int. Conf. Learn Represent. (ICLR), 2019.

[29] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, "Hierarchical graph representation learning withdifferentiable pooling," 2018, arXiv:1806.08804.

[30] H. Gao and S. Ji, "Graph u-nets," 2019. [Online]. Available: http://arxiv.org/abs/1905.05178

[31] F. M. Bianchi, D. Grattarola, and C. Alippi, "Spectral clustering with graph neural networks for graph pooling," in *Proc. 37th Int. Conf. Machine Learn.*, ACM, 2020.

[32] C. Cangea, P. Velicković, N. Jovanović, T. Kipf, and P. Liò, "Towards sparse hierarchical graph classifiers," 2018, arXiv:1811.01287.

[33] F. Chollet et al., "Keras." 2015. https://keras.io

[34] M. Abadi et al., "Tensorflow: A system for large-scale machine learning," in *Proc. 12th {USENIX} Symp. Oper. Syst. Design Implement. ({OSDI} 16)*, 2016, pp. 265–283.

[35] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting," 2017, arXiv:1707.01926.

[36] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2015, arXiv:1511.05493.

[37] K. K. Thekumparampil, C. Wang, S. Oh, and L.-J. Li, "Attention-based graph neural network for semi-supervised learning," 2018, arXiv:1803.03735.

[38] J. Du, S. Zhang, G. Wu, J. M. Moura, and S. Kar, "Topology adaptive graph convolutional networks," 2017, arXiv:1710.10370.

[39] T. Xie and J. C. Grossman, "Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties," *Phys. Rev. Lett.*, vol. 120, no. 14, p. 145301, 2018. doi: 10.1103/PhysRevLett.120.145301.

[40] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds.(2018)," 2018, arXiv:1801.07829.

[41] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," 2019. [Online]. Available: http://arxiv.org/abs/1904.08082

[42] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Proc. AAAI Conf. Artific. Intell.*, 2018.

[43] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, "Collective classification in network data," *AI Mag.*, vol. 29, no. 3, pp. 93–93, 2008. doi: 10.1609/aimag.v29i3.2157.

[44] C. Stark, B.-J. Breitkreutz, T. Reguly, L. Boucher, A. Breitkreutz, and M. Tyers, "Biogrid: a general repository for interaction datasets," *Nucleic Acids Res.*, vol. 34, no. suppl_1, pp. D535–D539, 2006. doi: 10.1093/nar/gkj109.

[45] M. Zitnik and J. Leskovec, "Predicting multicellular function through multi-layer tissue networks," *Bioinformatics*, vol. 33, no. 14, pp. i190–i198, 2017. doi: 10.1093/bioinformatics/btx252.

[46] R. Ramakrishnan, P. O. Dral, M. Rupp, and O. A. Von Lilienfeld, "Quantum chemistry structures and properties of 134 kilo molecules," *Sci. Data*, vol. 1, p. 140022, 2014. doi: 10.1038/sdata.2014.22.

[47] K. Kersting, N. M. Kriege, C. Morris, P. Mutzel, and M. Neumann, "Benchmark data sets for graph kernels," 2016. [Online]. Available: https://ls11-www.cs.tu-dortmund.de/staff/morris/graphkerneldatasets

[48] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," 2019, arXiv:1903.02428.

[49] M. Wang et al., "Deep graph library: Towards efficient and scalable deep learning on graphs," 2019, arXiv:1909.01315.

[50] Keras. "Why use Keras." 2019. https://keras.io/why-use-keras/

[51] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann, "Pitfalls of graph neural network evaluation," 2018, arXiv:1811.05868.

[52] J. Klicpera, J. Groß, and S. Günnemann, "Directional message passing for molecular graphs," 2020, arXiv:2003.03123.

[53] S. Ivanov, S. Sviridov, and E. Burnaev, "Understanding isomorphism bias in graph data sets," 2019.