Graph Neural Networks in TensorFlow and Keras with Spektral

Daniele Grattarola ¹ Cesare Alippi ²

Abstract

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.

1. Introduction

Graph Neural Networks (GNNs) are a class of deep learning methods designed to perform inference on data described by graphs (Battaglia et al., 2018). 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 tasks. In physics, GNNs have been used to learn physical models of complex systems of interacting particles (Battaglia et al., 2016; Kipf et al., 2018; Sanchez-Gonzalez et al., 2018; Farrell et al., 2018). 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 (Berg et al., 2017; Ying et al., 2018a). GNNs have also been largely applied to the biological sciences, with applications ranging from the recommendation of medications (Shang et al., 2019), to the prediction of protein-protein and protein-ligand interactions (Gainza et al., 2020), and in chemistry, for the prediction of quantum molecular properties as well as the generation of novel compounds and drugs (Do et al., 2019; You et al., 2018). Finally, GNNs have been successfully applied in fields like natural language processing (Fernandes et al., 2018; De Cao et al., 2018) and even more complex tasks like abstract reasoning (Santoro et al., 2017; Allamanis et al., 2017; Schlichtkrull et al., 2018) and decision making with reinforcement learning (Zambaldi et al., 2018; Hamrick et al., 2018).

At the core of GNNs there are two main types of operations, which can be interpreted as a generalisation of the convolution and pooling operators in convolutional neural networks: message passing and graph pooling (Fig. 1). 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 (Gilmer et al., 2017; Scarselli et al., 2009; Defferrard et al., 2016; Kipf & Welling, 2016; Simonovsky & Komodakis, 2017; Velickovic et al., 2017; Hamilton et al., 2017; Bianchi et al., 2019; Xu et al., 2019; Klüner et al., 2019) or graph pooling (Ying et al., 2018b; Gao & Ji, 2019; Klüner et al., 2019) or graph pooling (Ying et al., 2018b; Gao & Ji, 2019; Bianchi et al., 2020; Cangea et al., 2018).

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” (Chollet et al., 2015). 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 (Abadi et al., 2016) 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 characterises 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 an 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.

---

\(^1\)https://graphneural.network
\(^2\)https://github.com/danielegrattarola/spektral

\(^3\)The name \textit{convolutional} derives from the homonymous module in Keras, as well as message-passing layers being originally derived as a generalisation of convolutional operators.
computed through a simple attentional mechanism weighted sum of the node features, where the weights are computed through a simple attentional mechanism.

2.3. 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. Because 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 (Ying et al., 2018b), MinCut pooling (Bianchi et al., 2020), TopK pooling (Cangea et al., 2018; Gao & Ji, 2019), and Self-Attention Graph Pooling (SAGPool) (Lee et al., 2019). 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) (Li et al., 2015), SortPool (Zhang et al., 2018), and attention-weighted sum (AWSP).

2.4. 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 (Sen et al., 2008); the protein-protein interaction dataset (PPI) (Stark et al., 2006; Zitnik & Leskovec, 2017; Hamilton et al., 2017) and the Reddit communities network dataset (Hamilton et al., 2017) from the GraphSAGE paper (Hamilton et al., 2017); the QM9 chemical dataset of small molecules (Ramakrishnan et al., 2014); the MNIST 8-NN...

### Table 1. Comparison of different GNN libraries.

<table>
<thead>
<tr>
<th>Library</th>
<th>Framework</th>
<th>MP</th>
<th>Pooling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spektral</td>
<td>TensorFlow</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>PyG</td>
<td>PyTorch</td>
<td>28</td>
<td>14</td>
</tr>
<tr>
<td>DGL</td>
<td>PyTorch, others</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>StellarGraph</td>
<td>TensorFlow</td>
<td>6</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Graph Neural Networks in TensorFlow and Keras with Spektral

2.5. Other tools

Some of the secondary functionalities implemented by Spektral include: the `utils` module, which exposes some useful utilities for graph deep learning (e.g., methods for computing the characteristic graph matrices or manipulating the data); the `chem` module, which offers tools for loading and processing molecular graphs; the `layers.ops` module, which offers a set of common operations that can be used by advanced users to create new GNN layers, like wrappers for common matrix operations that automatically handle sparse inputs and batches of graphs.

3. 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 (PyG) (Fey & Lenssen, 2019) and the Deep Graph Library (DGL) (Wang et al., 2019), both of which are based on the PyTorch deep learning library. Instead, Spektral is specifically developed for the TensorFlow ecosystem, which to this date is estimated to support the majority of deep learning applications both in research and industry (Keras, 2019). The features offered by Spektral, summarized in Table 1, 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 computational performance of Spektral’s layers is also comparable to that of PyG, with small differences due to implementation details and differences between the libraries.
two respective backend frameworks. We also mention 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.

4. 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 C.

Node classification In our first experiment, we consider a task of semi-supervised node classification on the Cora, CiteSeer, and Pubmed citation networks. We evaluate GCN (Kipf & Welling, 2016), ChebNets (Defferrard et al., 2016), ARMA (Bianchi et al., 2019), GAT (Velickovic et al., 2017) and APPNP (Klicpera et al., 2019). We reproduce the same experimental settings described in the original papers, but we use the random data splits suggested by Shchur et al. (2018) for a fairer evaluation.

Graph classification To evaluate the pooling layers, DiffPool, MinCut, Top-K, and SAGPool, 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 C) 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).

Graph regression To evaluate the global pooling methods, we use the QM9 molecular database and train a GNN on four different regression targets: dipole moment (Mu), isotropic polarizability (Alpha), energy of HOMO (Homo), and internal energy at OK (U0). Because 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.

4.1. Results

The results for each experiment are reported in Tables 2, 3, and 4. In the first experiment, results are compatible with what reported in the literature, although some differences in performance are present due to using a random data split rather than the pre-defined 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 standardized 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 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.

5. 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.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>ChebNets</th>
<th>GCN</th>
<th>GAT</th>
<th>ARMA</th>
<th>APPNP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cora</td>
<td>77.4 ± 1.5</td>
<td>79.4 ± 1.3</td>
<td>82.0 ± 1.2</td>
<td>80.5 ± 1.2</td>
<td>82.8 ± 0.9</td>
</tr>
<tr>
<td>CiteSeer</td>
<td>68.2 ± 1.6</td>
<td>68.8 ± 1.4</td>
<td>70.0 ± 1.0</td>
<td>70.6 ± 0.9</td>
<td>70.0 ± 1.0</td>
</tr>
<tr>
<td>Pubmed</td>
<td>74.0 ± 2.7</td>
<td>76.6 ± 2.5</td>
<td>73.8 ± 3.3</td>
<td>77.2 ± 1.6</td>
<td>78.2 ± 2.1</td>
</tr>
</tbody>
</table>

Table 2. Classification accuracy on the node classification tasks.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Flat</th>
<th>MinCut</th>
<th>DiffPool</th>
<th>Top-K</th>
<th>SAGPool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proteins</td>
<td>74.3 ± 4.5</td>
<td>75.5 ± 2.9</td>
<td>74.1 ± 3.9</td>
<td>70.5 ± 3.4</td>
<td>71.3 ± 5.0</td>
</tr>
<tr>
<td>IMDB-B</td>
<td>72.8 ± 7.2</td>
<td>73.6 ± 5.4</td>
<td>70.6 ± 6.4</td>
<td>67.7 ± 8.2</td>
<td>69.3 ± 5.7</td>
</tr>
<tr>
<td>Mutag</td>
<td>72.5 ± 14.0</td>
<td>81.4 ± 10.7</td>
<td>83.5 ± 29.7</td>
<td>79.2 ± 38.0</td>
<td>78.5 ± 28.3</td>
</tr>
<tr>
<td>NCI1</td>
<td>77.3 ± 2.6</td>
<td>74.4 ± 1.9</td>
<td>71.1 ± 3.0</td>
<td>72.0 ± 3.0</td>
<td>69.4 ± 28.4</td>
</tr>
</tbody>
</table>

Table 3. Classification accuracy on the graph classification tasks.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Average</th>
<th>Sum</th>
<th>Max</th>
<th>GAP</th>
<th>AWSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mu</td>
<td>1.12 ± 2.03</td>
<td>1.02 ± 0.02</td>
<td>0.90 ± 0.04</td>
<td>1.04 ± 0.05</td>
<td>0.99 ± 0.03</td>
</tr>
<tr>
<td>Alpha</td>
<td>3.15 ± 2.65</td>
<td>2.38 ± 0.64</td>
<td>6.20 ± 3.33</td>
<td>1.89 ± 2.09</td>
<td>31.1 ± 37.1</td>
</tr>
<tr>
<td>Homo</td>
<td>9.24 ± 2.41</td>
<td>9.22 ± 0.51</td>
<td>8.90 ± 2.06</td>
<td>9.04 ± 2.09</td>
<td>8.05 ± 2.09</td>
</tr>
<tr>
<td>U0</td>
<td>0.42 ± 0.14</td>
<td>0.50 ± 0.13</td>
<td>11.07 ± 4.5</td>
<td>0.22 ± 0.13</td>
<td>624.0 ± 19.0</td>
</tr>
</tbody>
</table>

Table 4. Mean-squared error on the graph regression tasks. Results for Homo are in scale of 10⁻⁵.
References


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


Graph Neural Networks in TensorFlow and Keras with Spektral


A. Data modes
Spektral 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 graphs in a batch. Alternatively, in batch mode, a set of graphs is represented by stacking their adjacency and attributes matrices in higher-order tensors of shape $B \times N \times \ldots$. 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.

In Spektral, all layers implementing message-passing operations are compatible with single and disjoint mode, and more than half of the layers also support mixed and batch mode. A similar consideration holds for pooling methods where however, due to structural limits of the methods themselves, some data modes cannot be supported directly.

B. Technical notes
Spektral is distributed through the Python Package Index (package name: spektral), supports all UNIX-like platforms, 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 stand-alone 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.

C. Experimental details
This section summarises the architectures and hyperparameters used in the experiments of Section 4. All experiments were run on a single NVIDIA Titan V GPU with 12GB of video memory.

C.1. 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 (Shchur et al., 2018), we use random splits with 20 labels per class for training, 30 labels per class for early stopping, all the remaining labels for testing.

C.2. 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 (Bianchi et al., 2019), 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 regularization 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;
Graph Neural Networks in TensorFlow and Keras with Spektral

• 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 (Ivanov et al., 2019), 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.3. Graph regression

We configure a GNN with the following structure: ECC - ECC - GLOBALPOOLING - DENSE, where ECC indicates an Edge-Conditioned Convolutional layer (Simonovsky & Komodakis, 2017) and GLOBALPOOLING indicates the global pooling layer being tested. ECC layers have 32 units each, and ReLU activation. No regularization 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 (Ramakrishnan et al., 2014).

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.