Learning to Aggregate on Structured Data

Master Thesis Proposal & Work Plan

Clemens Damke

Matriculation Number: 7011488

cdamke@mail.uni-paderborn.de

October 24, 2019

1 Motivation

Most of the commonly used supervised machine learning techniques assume that instances are represented by d-dimensional feature vectors $x_i \in \mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_d$ for which some target value $y_i \in \mathcal{Y}$ should be predicted. In the regression setting the target domain $\mathcal{Y}$ is continuous, typically $\mathcal{Y} = \mathbb{R}$, whereas $\mathcal{Y}$ is some discrete set of classes in the classification setting.

Since not all data is well-suited for a fixed-dimensional vector representation, approaches that directly consider the structure of the input data might be more appropriate in such cases. One such case is the class of so-called *learning to aggregate* (LTA) problems as described by Melnikov and Hüllermeier [1]. There the instances are represented by compositions $\boldsymbol{c}_i$ of constituents $c_{i,j} \in \boldsymbol{c}_i$, i.e. variable-size multisets with $n_i = |\boldsymbol{c}_i|$. The assumption in LTA problems is that for all constituents $c_{i,j}$ a local score $y_{i,j} \in \mathcal{Y}$ is either given or computable. The set of those local scores should be indicative of the overall score $y_i \in \mathcal{Y}$ of the composition $\boldsymbol{c}_i$. LTA problems typically require two subproblems to be solved:

1. **Aggregation:** A variadic aggregation function $A : \mathcal{Y}^* \to \mathcal{Y}$ that estimates composite scores has to be learned, i.e. $y_i \approx \hat{y}_i = A(y_{i,1}, \dots, y_{i,n_i})$. Typically

1

the aggregation function A should be associative and commutative to fit with the multiset-structure of compositions.

2. **Disaggregation:** In case the constituent scores $y_{i,j}$ are not given, they have to be derived from a constituent representation, e.g. a vector $v_{i,j} \in \mathcal{V}$. To learn this derivation function $f : \mathcal{V} \to \mathcal{Y}$, only the constituent vectors $\{v_{i,j}\}_{j=1}^{n_i}$ and the composite score y_i is given. Thus the constituent scores $y_{i,j}$ need to be *disaggregated* from y_i in order to learn f.

Overall LTA can be understood as the joint problem of learning the aggregation function A and the local score derivation function f.

Current LTA approaches only work with multiset inputs. In practice there is however often some relational structure among the constituents of a composition. This effectively turns LTA into a graph regression problem. The goal of this thesis is to look into the question of how aggregation function learning methods might be generalized to the graph setting.

2 Related Work

This thesis will be based on two currently mostly separate fields of research: **1.** Learning to Aggregate **2.** Graph classification & regression. A short overview of the current state-of-the-art approaches in both fields will be given now.

2.1 Learning to Aggregate

Two main approaches to represent the aggregation function in LTA problems have been explored. The first approach uses *uninorms* [1] to do so. There the basic idea is to express composite scores as fuzzy truth assignments $y_i \in [0, 1]$. Such a composite assignment y_i is modeled as the result of a parameterized logical expression of constituent assignments $y_{i,j} \in [0, 1]$. As the logical expression that thus effectively aggregates the constituents, a uninorm U_λ is used. Depending on the parameter λ, U_λ combines t-norms and t-conorms which are continuous generalizations of logical conjunction and disjunction respectively.

Recently Melnikov and Hüllermeier [2] have also looked at an alternative class of aggregation functions. Instead of using fuzzy logic to describe score aggregation,

ordered weighted average (OWA) operators were used. OWA aggregators work by sorting the input scores and then weighting them based on their sort position, i.e.

$$A_\lambda(y_1, \ldots, y_n) := \sum_{i=1}^{n} \lambda_i y_{\pi(i)},$$

where λ is a weight vector with $\|\lambda\|_1 = 1$ and π is a sorting permutation of the input scores. To deal with varying composite sizes n the weights λ_i are interpolated using a *basic unit interval monotone* (BUM) function $q : [0, 1] \to [0, 1]$. It takes constituent positions that are normalized to the unit interval, i.e. $\frac{i}{n}$. The BUM function q then is then used to interpolate a weight for any normalized sort position via $\lambda_i = q(\frac{i}{n}) - q(\frac{i-1}{n})$. Therefore the learning objective boils down to optimizing the shape of q. The details of this are left out here.

2.2 Graph Classification and Regression

As previously mentioned, the addition of relations between constituents turns LTA into a graph regression problem. Most of the recent research in the field of learning from graph structured data has however focused on the closely related graph classification problem. Since many ideas from the classification setting are also useful in the regression setting, a brief overview of those ideas is given first.

At a high level graph classification methods can be taxonomized into two main families:

1. **Vector representation approaches:** One way to tackle the graph classification problem is to map an input graph G to a vectorial representation. This can be done a) by either handpicking global graph features like vertex/edge count, degree distribution or graph diameter, b) via a graph embedding algorithm like node2vec [3], sub2vec [4] or graph2vec [5], c) implicitly by using a graph kernel that computes the structural similarity between graphs, e.g. the Weisfeiler-Lehman kernel [6] or the multiscale Laplacian graph kernel [7]. Graphs can then be classified via any classification algorithm that works with vectors and/or kernels.

2. **Graph neural networks:** An alternative approach is to adapt neural networks to graph inputs. The notion of a *graph neural network* (GNN) was first introduced by Gori et al. [8]. There a message-passing architecture is

used to iteratively propagate vertex information to neighbors until a fixed point is reached. This process is computationally expensive.

Recently the class of *convolutional graph neural networks* (ConvGNNs) gained traction. It generalizes the convolution operator that is used by *convolutional neural networks* (CNNs) to graphs. There are two main variants of ConvGNNs:

a) **Spatial variant:** Standard CNNs use a convolution operator that is typically applied to a grid graph of pixels with added diagonal edges. Spatial ConvGNNs [9] directly generalize this idea by defining a more flexible convolution that works with arbitrary graph structures. Spatial graph convolution aggregates the direct neighborhood of a vertex v_i, where v_i is typically described by a feature vector $x_i \in \mathbb{R}^d$. This process returns a new aggregate feature vector x_i'. By stacking multiple convolution layers, features from indirect neighbors become part of the aggregate. Analogous to how CNNs learn kernel matrices, spatial ConvGNNs learn vertex neighborhood feature aggregation functions. This approach shares similarities with the previously mentioned message-passing architecture [8] with the major different being that the number of message-passing, i.e. convolution, steps is fixed.

b) **Spectral variant:** Motivated by the theoretical foundation provided by spectral graph theory [10], spectral ConvGNNs [11] learn a filter function $\hat{g}$ that is applied in the Fourier domain of a graph. Based on the convolutional theorem, a convolution operator can not only be expressed in terms of the neighborhood of vertices but also as a filter on the eigenvectors of a graph's Laplacian. Formally this can be expressed as

$$g *_G \boldsymbol{x} = U\hat{g}(\Lambda)U^\top \boldsymbol{x} \tag{1}$$

where $\boldsymbol{x} \in \mathbb{R}^{n \times k}$ is the matrix of all vertex features and $U\Lambda U^\top$ is the eigendecomposition of the graph Laplacian L_G[1]. In the previously described spatial ConvGNN variant, g is learned directly in form of the feature aggregation function with a neighborhood locality constraint. In the spectral variant however the Fourier-transformed filter $\hat{g}$ is learned. Intuitively this means that vertex features x_i are not aggregated with

---

[1] To see the connection to the convolutional theorem, note that $\mathcal{F} = U$ and $\mathcal{F}^{-1} = U^\top$ in eq. (1) with $\mathcal{F}$ denoting the graph Fourier transform.

the features of their their direct neighborhood but with the features of vertices with which they share certain structural similarities. This allows spectral ConvGNNs to incorporate the global graph structure at the cost of having to perform the computationally expensive decomposition of every input graph's Laplacian.

To tackle the performance impact of the spectral convolution approach, various simplifications of the filter $\hat{g}$ have been proposed. The so-called *graph convolutional network* (GCN) [12] does this by reducing the expressivity of the filter. Then $\hat{g}$ can be applied to the Laplacian directly via $g *_G \boldsymbol{x} = \hat{g}(L_G)\boldsymbol{x}$ which saves the decomposition costs. This simplification of $\hat{g}$ implicitly causes its Fourier inverse g to be locality constrained just like in the spatial ConvGNN variant. Hence the spectral GCN approach also allows for a spatial/message-passing interpretation. Recently various variants and extensions of GCNs [13][14][15][16][17] have been proposed that apply additional approximations and sampling methods to improve the runtime as well as the accuracy in certain scenarios.

One notable extension of GCNs are the so-called *adadptive graph convolutional networks* (AGCNs) [18]. AGCNs do not use a fixed Laplacian L for each input but instead learn a so-called residual Laplacian L_{res}. The residual Laplacian is constructed by learning a vertex similarity kernel $\mathbb{G}_{x_i,x_j}$ which is used to determine edge weights between vertices. By overlaying the learned residual Laplacian L_{res} on top of the given intrinsic Laplacian L, similar nodes with a large spatial distance in L become connected. This can improve the accuracy especially if a locality constrained spectral filter $\hat{g}$ is used.

The just described ConvGNN approaches generally assume that an input consists of an undirected graph with positive real edge weights. Those approaches are not directly applicable in scenarios with multiple edge types, i.e. multi-relational scenarios. One class of spatial multi-relational GNN methods was proposed by Battaglia et al. [19]. There a shared function ϕ^e is used to describe how a vertex feature vector v_i is transformed as it passes through an edge with features $e_{ij} \in \mathbb{R}^p$. In a uni-relational spatial GNN with $e_{ij} \in \mathbb{R}^1$, typically $\phi^e(e_{ij}, v_i) = e_k \cdot v_i$ is used. In the multi-relational setting, ϕ^e can instead be learned via a standard *multilayer perceptron* (MLP). An

alternative spectral multi-relational ConvGNN approach was proposed by Gong and Cheng [20]. Their so-called *edge enhanced graph neural network* (EGNN) essentially models each relation as a separate graph with its own Laplacian. A convolutional filter *g* is shared across all relations and used to convolve the input signal separately via each relation's Laplacian. To share vertex features across relations, the separately convolved signals of all relations are then concatenated. Those concatenated vertex features are used as the input signal of the next layer.

This concludes the short overview of approaches to learn vertex feature vectors via ConvGNNs. After the application of either spatial or spectral graph convolutions, each vertex will have an updated aggregate feature vector. To obtain an aggregate score for the entire graph, a graph pooling layer is applied. A primitive pooling approach is to simply merge the vertex features via a fixed aggregation function like min, max or avg. Better results can be obtained by learning the pooling aggregation function. Two notable graph pooling approaches are *SortPooling* [21] and *Self-Attention Pooling* [22].

a) SortPooling assumes that the vertex feature vectors are obtained using a stack of spectral convolution layers that learn filters on the random walk Laplacian $L = I - D^{-1}A$. This assumption makes it possible to interpret the outputs of the graph convolution layers as a continuous generalization of *Weisfeiler-Lehman* (WL) colors. WL colors are lexicographically sortable signatures that encode the structural role of vertices; they can be used for efficient graph isomorphism falsification [23]. Using this interpretation of the vertex feature vectors, SortPooling determines a top-k ranking of the vertices' WL colors and then aggregates the thus selected fixed-size vertex feature subset via a MLP.

b) Self-Attention Pooling adds an additional graph convolution layer after an arbitrary ConvGNN. The added layer takes the aggregated vertex feature vectors as input and outputs a score for each vertex. This score is used to determine a top-k vertex ranking. Unlike SortPooling the feature vectors of the resulting subset are first added up. The resulting aggregate graph feature vector is then fed into a MLP to obtain a class.

While all of the previously mentioned problems were evaluated on graph classification problems, they can naturally be extended to regression tasks. In the case of ConvGNNs for example, the described pooling layers already produce continuous

outputs.

The pooling aggregation functions used in ConvGNNs are however typically not part of the learning objective. This shows the fundamental methodological difference between GNNs and LTA. A GNN uses the relational structure of its input to try to determine meaningful feature representations or scores for each vertex. Those vertex scores are then combined using a simple aggregation function. LTA on the other hand currently does not use any relational structure and therefore computes vertex/constituent scores independently. Those comparatively less meaningful local scores are then combined using a more expressive learned aggregation function. Generally speaking current GNN approaches focus on the disaggregation problem while LTA approaches focus on the aggregation problem. The described Self-Attention Pooling [22] can be seen as an exception to this generalization since it learns an aggregation function in form of the self-attention vertex weights.

2.3 Molecular Analysis Problems and Datasets

Graph regression problems have for the most part only been discussed in the context of specific domain problems. One of those domains is the prediction of chemical toxicity, typically the LD_{50} or LC_{50} doses which measure the smallest amount of a given chemical that is lethal for at least 50% of tested subjects. Another regression task from the domain of molecular analysis is the prediction of compound solubility. The following approaches could serve as inspiration, baselines and/or sources of training data:

- *RASAR* [24] uses a logistic regression model. It is trained with manually chosen features that are given for training molecule graphs. To predict the toxicity of a new chemical, k-means is used to average the features of chemicals with similar molecule graphs in the training set. The structural similarity between molecule graphs is measured by the Tanimoto coefficient of their chemical fingerprints. Those fingerprints encode the presence or absence of certain meaningful chemical substructures. The implementation and used dataset is unfortunately not publicly available because it is part of a commercial product[2].

- *ProTox* [25][26] is another toxicity prediction approach. It works similarly to RASAR but leaves out the logistic regression step and instead directly

---

[2]ToxTrack Inc., `https://toxtrack.com/`

aggregates the toxicity of similar chemicals via k-means. ProTox uses the freely available *SuperToxic* dataset [27] which contains 13400 molecules with known LD_{50} values. While the implementation is again not publicly available, the trained model can be freely queried online[3].

- The *Toxicity Estimation Software Tools* (T.E.S.T.)[4] is a free collection of models for toxicity prediction. Like RASAR and ProTox it includes similarity-based prediction models but also linear regression and decision tree models that are trained on vector representations of chemicals. The publicly available *ChemIDplus*[5] database was used for training. It contains 28023 and 13541 molecules with known oral LD_{50} values for rats and mice respectively. Additionally about 1000 molecules with known LC_{50} values are available for both species.

- Recently first toxicity prediction approaches using ConvGNNs have been published. The previously mentioned AGCN [18] was in fact evaluated on a discrete toxicity classification problem; this could serve as a starting point for a continuous graph regression method. Pope et al. [28] describe how functional groups of toxic chemicals can be learned using GCNs; there a functional group refers to a relevant subgraph of a molecule that can be interpreted as a toxicity indicator.

- The previously described EGNN method [20] for multi-relational graphs was evaluated on a compound solubility prediction task. For this the Lipophilicity [29] and FreeSolv [30] datasets were used. They contain 4200 and 642 labeled molecular graphs respectively.

3 Goals

As mentioned in section 1, the overall goal of the thesis is to extend the LTA approach described in section 2.1 to the graph setting. Since this goal strongly overlaps with the work on graph classification described in section 2.2, both branches of research should be combined. The practical applications described in section 2.3 could be used for comparison and as a potential source of graph

---

[3]ProTox-II, http://tox.charite.de/protox_II/

[4]https://www.epa.gov/chemical-research/toxicity-estimation-software-tool-test

[5]https://chem.nlm.nih.gov/chemidplus/

regression training/test datasets. Now follows a concrete list of required and optional goals that should be achieved.

3.1 Required Goals

1. **Aggregation:** Define a class of aggregation functions that takes vertex scores as its input. It should fulfill the role of the uninorms and OWA functions in the existing LTA papers (see section 2.1).

2. **Disaggregation:** Describe how vertex features should be used to compute a constituent score that will be combined by the aggregation function.

3. **Learning:** Combine the chosen aggregation and disaggregation methods into a joint LTA model with a suitable loss function.

4. **Implementation:** A reference implementation for the designed LTA model should be developed.

5. **Evaluation:** The reference implementation should be used to evaluate the model on at least one test dataset. Due to the relative scarcity of different graph regression datasets, the used dataset might also be synthetically generated. A generated dataset should be obtained using a variety of representative deterministic graph scoring measures to check which kinds of local and global structural features are learnable; some examples of measures that might be part of a graph scoring functions are the count of k-cliques in a graph, the graph diameter, the presence of cycles or various properties of a graph's degree distribution. The results should be compared with the results of at least one other approach for learning graph scores, e.g. a ConvCNN with a fixed aggregation function instead of a learned one.

3.2 Optional Goals

If time permits, the previous goals can be extended. For each of the previous five goal categories, optional additional ideas are listed below.

1. **Aggregation:** Multiple classes of aggregation functions could be considered and compared.

2. **Disaggregation:** As described in section 2.2 there are multiple methods to learn vertex scores. Depending on the chosen overall approach, a few of those methods might be suitable to be combined with an aggregation function learner. Those suitable combinations could be evaluated and compared.

3. **Learning:** The joint model of aggregation and disaggregation function learning can most likely be optimized using various algorithms. Those algorithms could be compared. In case their runtime performance is impractical in certain situations, approximations and simplifications of the model could be considered.

4. **Implementation:** The chosen approach should ideally align well with the practical constraints of implementation. It should be suitable for an implementation on modern GPUs and/or distributed cluster environments.

5. **Evaluation:** If possible, real datasets should be used for evaluation (see section 2.3) as an alternative or in addition to generated data. Aspects like runtime performance and scalability could also be evaluated.

4 Approach

As described in section 1, to extend LTA to graph inputs the aggregation and disaggregation problems have to be solved. Since the various approaches to learn vertex features described in section 2.2 can already be interpreted as solutions for the disaggregation problem, only a compatible vertex score aggregation function learning method has to be developed. This corresponds to learning a pooling function in ConvGNNs. The previously mentioned SortPooling and Self-Attention Pooling methods already do this in the classification setting. A method for the regression setting has to be found.

To do so, training and test data is needed in the first step. Therefore one or multiple of the chemical toxicity datasets mentioned in section 2.3 should be prepared for the regression pipeline. This includes the following tasks: 1. The toxicity data has to be read from various formats (CSV, JSON or XML from a REST-API, chemical SD files, etc.), 2. serialized SMILES strings of molecular structure need to be translated into adjacency matrices[6] and 3. the various units of measurement for toxicity

---

[6]e.g. using RDKit (https://github.com/rdkit/rdkit), pysmiles (https://github.com/pckroon/pysmiles) or Open Babel (http://openbabel.org)

need to be normalized. In case the toxicity data turns out to be unsuitable, an artificial dataset might alternatively have to be generated as described in section 3.1. Whether this is necessary will only become apparent after the first evaluations which could show that the chosen approach is completely ill suited to the toxicity prediction task.

In the next step the development of a vertex aggregation method can begin. For the disaggregation problem the open source reference implementation of GCNs[7] or more generally the Graph Nets library[8] for TensorFlow might be useful. The development should start by building a simple ConvGNN with a fixed aggregation function. Then an aggregation function learner can be added. Since the existing GNN tools and reference implementations are mostly available in Python, the implementation part of this thesis should probably also use this language.

Lastly the developed method should be evaluated using the dataset that was chosen in the first step. The results on the toxicity dataset should be compared with the results of selected approaches from section 2.3. Additionally the learned aggregation functions should be compared with naïve aggregation functions like min, max or avg since those aggregation functions are typically used in the pooling layer of current ConvGNN approaches.

5 Preliminary Document Structure

1. Introduction

2. Previous Work

    2.1. Learning to Aggregate

    2.2. Convolutional Graph Neural Networks

3. Learning to Aggregate Graphs

    3.1. A Vertex Aggregation Function Learner

    3.2. Compatible Disaggregation Methods

4. Evaluation

---

[7] https://github.com/tkipf/gcn
[8] https://github.com/deepmind/graph_nets
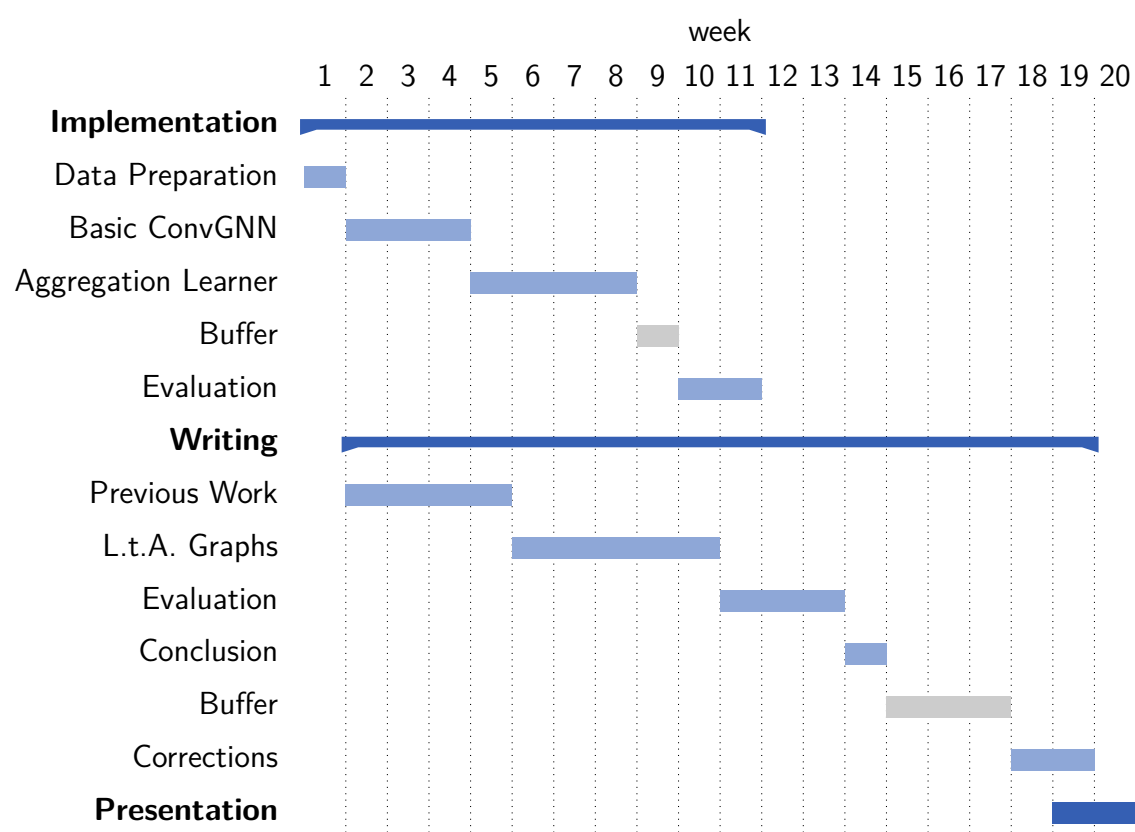
6 Time-Schedule

Figure 1: Sketch of the time schedule for the work on the thesis

12

References

[1] Vitalik Melnikov and Eyke Hüllermeier. "Learning to Aggregate Using Uninorms." In: *Machine Learning and Knowledge Discovery in Databases.* Springer International Publishing, 2016, pp. 756–771 (cit. on pp. 1, 2).

[2] Vitalik Melnikov and Eyke Hüllermeier. "Learning to Aggregate: Tackling the Aggregation/Disaggregation Problem for OWA." In: *ACML* (2019) (cit. on p. 2).

[3] Aditya Grover and Jure Leskovec. "node2vec: Scalable Feature Learning for Networks." In: (July 3, 2016). arXiv: `1607.00653v1` [`cs.SI`] (cit. on p. 3).

[4] Bijaya Adhikari, Yao Zhang, Naren Ramakrishnan, and B. Aditya Prakash. "Sub2Vec: Feature Learning for Subgraphs." In: *Advances in Knowledge Discovery and Data Mining.* Springer International Publishing, 2018, pp. 170–182 (cit. on p. 3).

[5] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, et al. "graph2vec: Learning Distributed Representations of Graphs." In: (July 17, 2017). arXiv: `1707.05005v1` [`cs.AI`] (cit. on p. 3).

[6] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. *Weisfeiler-Lehman Graph Kernels.* 2011 (cit. on p. 3).

[7] Risi Kondor and Horace Pan. "The Multiscale Laplacian Graph Kernel." In: (Mar. 20, 2016). arXiv: `1603.06186v2` [`stat.ML`] (cit. on p. 3).

[8] M. Gori, G. Monfardini, and F. Scarselli. "A new model for learning in graph domains." In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* IEEE, 2005 (cit. on pp. 3, 4).

[9] A. Micheli. "Neural Network for Graphs: A Contextual Constructive Approach." In: *IEEE Transactions on Neural Networks* 20.3 (2009), pp. 498–511 (cit. on p. 4).

[10] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst. "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains." In: *IEEE Signal Processing Magazine* 30.3 (2013), pp. 83–98 (cit. on p. 4).

[11] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. "Spectral Networks and Locally Connected Networks on Graphs." In: (Dec. 21, 2013). arXiv: `1312.6203v3` [`cs.LG`] (cit. on p. 4).

[12] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks." In: *ICLR* (Sept. 9, 2016). arXiv: `1609.02907v4 [cs.LG]` (cit. on p. 5).

[13] William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs." In: (June 7, 2017). arXiv: `1706.02216v4 [cs.SI]` (cit. on p. 5).

[14] Jie Chen, Tengfei Ma, and Cao Xiao. "FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling." In: (Jan. 30, 2018). arXiv: `1801.10247v1 [cs.LG]` (cit. on p. 5).

[15] Jianfei Chen, Jun Zhu, and Le Song. "Stochastic Training of Graph Convolutional Networks with Variance Reduction." In: (Oct. 29, 2017). arXiv: `1710.10568v3 [stat.ML]` (cit. on p. 5).

[16] Wei-Lin Chiang, Xuanqing Liu, Si Si, et al. "Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks." In: (May 20, 2019). arXiv: `1905.07953v2 [cs.LG]` (cit. on p. 5).

[17] Jian Du, Shanghang Zhang, Guanhang Wu, Jose M. F. Moura, and Soummya Kar. "Topology Adaptive Graph Convolutional Networks." In: (Oct. 28, 2017). arXiv: `1710.10370v5 [cs.LG]` (cit. on p. 5).

[18] Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. "Adaptive Graph Convolutional Neural Networks." In: (Jan. 10, 2018). arXiv: `1801.03226v1 [cs.LG]` (cit. on pp. 5, 8).

[19] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, et al. "Relational inductive biases, deep learning, and graph networks." In: (June 4, 2018). arXiv: `http://arxiv.org/abs/1806.01261v3 [cs.LG]` (cit. on p. 5).

[20] Liyu Gong and Qiang Cheng. "Exploiting Edge Features in Graph Neural Networks." In: (Sept. 7, 2018). arXiv: `http://arxiv.org/abs/1809.02709v2 [cs.LG]` (cit. on pp. 6, 8).

[21] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. "An end-to-end deep learning architecture for graph classification." In: *Thirty-Second AAAI Conference on Artificial Intelligence.* 2018 (cit. on p. 6).

[22] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. "Self-Attention Graph Pooling." In: (Apr. 17, 2019). arXiv: `1904.08082v4 [cs.LG]` (cit. on pp. 6, 7).

[23] Boris Weisfeiler and Andrei A. Lehman. "A reduction of a graph to a canonical form and an algebra arising during this reduction." In: *Nauchno-Technicheskaya Informatsia* 2.9 (1968), pp. 12–16 (cit. on p. 6).

[24] Thomas Luechtefeld, Dan Marsh, Craig Rowlands, and Thomas Hartung. "Machine Learning of Toxicological Big Data Enables Read-Across Structure Activity Relationships (RASAR) Outperforming Animal Test Reproducibility." In: *Toxicological Sciences* 165.1 (2018), pp. 198–212 (cit. on p. 7).

[25] Malgorzata N. Drwal, Priyanka Banerjee, Mathias Dunkel, Martin R. Wettig, and Robert Preissner. "ProTox: a web server for the in silico prediction of rodent oral toxicity." In: *Nucleic Acids Research* 42.W1 (2014), W53–W58 (cit. on p. 7).

[26] Priyanka Banerjee, Andreas O. Eckert, Anna K. Schrey, and Robert Preissner. "ProTox-II: a webserver for the prediction of toxicity of chemicals." In: *Nucleic Acids Research* 46.W1 (2018), W257–W263 (cit. on p. 7).

[27] U. Schmidt, S. Struck, B. Gruening, et al. "SuperToxic: a comprehensive database of toxic compounds." In: *Nucleic Acids Research* 37.Database (2009), pp. D295–D299 (cit. on p. 8).

[28] Phillip Pope, Soheil Kolouri, Mohammad Rostrami, Charles Martin, and Heiko Hoffmann. "Discovering Molecular Functional Groups Using Graph Convolutional Neural Networks." In: (Dec. 1, 2018). arXiv: `1812.00265v3 [cs.LG]` (cit. on p. 8).

[29] Zhenqin Wu, Bharath Ramsundar, Evan N. Feinberg, et al. "MoleculeNet: A Benchmark for Molecular Machine Learning." In: (Mar. 2, 2017). arXiv: `http://arxiv.org/abs/1703.00564v3 [cs.LG]` (cit. on p. 8).

[30] David L. Mobley and J. Peter Guthrie. "FreeSolv: a database of experimental and calculated hydration free energies, with input files." In: *Journal of Computer-Aided Molecular Design* 28.7 (2014), pp. 711–720 (cit. on p. 8).

<div style="display:flex; justify-content:space-between;">
  <div>_____
Supervisor</div>
  <div>_____
Student</div>
</div>