PERFORMANCE EVALUATION OF PREDICTION ON MOLECULAR GRAPHS WITH GRAPH NEURAL NETWORKS

Haotian Li
Ludwig-Maximilians-Universität München
Beijing Normal University
H.Li1@campus.lmu.de

ABSTRACT

Machine learning and deep learning are novel and trending approaches to solving real-world scientific problems. Graph machine learning is dedicated to performing learning methods, such as graph neural networks, on non-Euclidean data such as graphs. Molecules, with their natural graph structures, could be analyzed by such method. In this work, we carry out the performance evaluation regarding to learning results as well as time consumed, speedup, and efficiency using different types of neural network structures and distributed training pipeline implementations. Besides, the reasons lead to an unideal performance enhancement is investigated. Code availability at `https://github.com/htlee6/perf-analysis-dist-training-gnn`.

***K*eywords** Graph Neural Network · Distributed Training · Molecular Graph

1 Introduction

Recent years have seen rapid development and success in the area of machine learning and deep learning. Machine learning and deep learning techniques have been widely applied in object detection, medical image analysis, natural language processing and understanding, audio recognition, recommendation systems, knowledge graphs, autonomous driving[1, 2, 3, 4, 5, 6, 7], etc. However, the sequentiality or the spacial order of the data limits the generalizability of these powerful models, for example, a convolutional neural network model cannot ingest a piece of non-Euclidean data. As a result, the problems of machine learning/deep learning on non-Euclidean data came into being, and the concept of graph machine learning, aiming at addressing this issue as a new approach, has been introduced.

In graph machine learning, the model is required to have the character of permutation invariance and equivariance. There are mainly 3 levels of tasks in the area of graph machine learning, i.e. node-level tasks, edge-level tasks, and graph-level tasks. Node-level tasks are concerned with predicting the identity or role of each node within a graph. An example of edge-level inference is predicting if an edge exists in the given graph, for example, when a new user joined Instagram, will the existing users follow the new user or vice versa? In a graph-level task, our goal is to predict the property of an entire graph. For example, for a molecule represented as a graph, we might want to predict whether it will bind to a receptor implicated in a disease.

In this work, we build a distributed training framework in order to investigate the effect of distributed training on the training performance (esp. time consumed), which is implemented using PyTorch[8] and PyTorch Ignite[9], and compare the performance difference between different graph machine learning libraries (PyG[10] and DGL[11]). From the results, we conclude that:

- Training a GIN model takes longer time than a GCN model, adding a virtual node in GNN model makes model more complex and therefore increases the training time.
- In our experiments, training with PyG is faster comparing to that using DGL with PyTorch as backend.
- The implementation of distributed training system did not achieve the expected performance, there are two known reasons: limited network communication rate and frequent switching between processes slowing the training down.

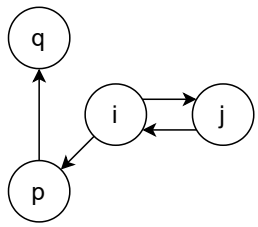

Figure 1: Example graph with 4 nodes, $i, j, p,$ and q

2 Learning on Molecular Graphs with Graph Neural Network

In this section, we are going to introduce some concepts, terminologies and models about molecular graph and graph neural network, as well as some popular libraries to build up a graph neural network model with ease.

2.1 Graph Representation and Molecular Graphs

Graph representation is often used in representing a graph object abstractly. A graph $G = (V, E)$, where $V = \{1, ..., n\}$ is the set of vertices, and $E \subseteq V \times V$ is the set of edges. One way to represent a graph G specifically is using an adjacency matrix $A \in R^{|V|\times|V|}$, for example, $A_{i,j} = 1$ means vertex i and j are connected in an undirected graph. Another way is to use an adjacency list L where the elements of the list are pairs of neighbor vertices, for example, $L = \{(i, j), (j, i), (p, q), (i, p)\}$ represents a directed graph G as figure 1 shows.

Molecular graphs are graphs that depict the topological structures of molecules. Chemically, the vertices of a molecular graph are the elements, such as C, H, O, etc., and the edges are chemical bonds that connect the elements. Besides the adjacency relationship as information contained in the graph, the vertices and edges could also contain special information in the given context, for example, the edges/chemical bonds of a molecular graph varies depending on the elements it is connected to, the edges of a knowledge graph varies depending on the given a tri-tuple $< s, r, d >$. Node features $\mathbf{X}^{(0)} \in \mathbb{R}^{|V|\times f}$ and edge features $\mathbf{E}^{(0)} \in \mathbb{R}^{|E|\times d}$ could be defined with regard to the domain knowledge, where f and d indicate the dimension of each feature.

2.2 Graph Neural Network

A graph neural network, namely GNN, is a learning module designed for learning on graph-structured data. A GNN iteratively updates node embeddings (eq. 2) by aggregating localized information (eq. 1) via the parameterized functions.

$$m_v^{(l+1)} = \mathrm{AGGREGATE}_{\theta_1}^{(l+1)} \left(\left\{ \left(x_w^{(l)}, x_v^{(l)}, e_{w,v}^{(l)} \right) \right\}_{w\in\mathcal{N}(v)} \right) \tag{1}$$

$$x_v^{(l+1)} = \mathrm{UPDATE}_{\theta_2}^{(l+1)} \left(x_v^{(l)}, m_v^{(l+1)} \right) \tag{2}$$

where $\{...\}$ denotes a multi-set and $\mathcal{N}(v) \in V$ defines the neighborhood set of node $v \in V$. [12] After L times of aggregation, a graph representation could be derived via global aggregation of $X^{(L)}$, such as summation, Set2set[13], or readout methods like virtual node[14] and etc. Figure 2 [15] illustrates the corresponding computational graph generated when calculating the embedding of target node A in a 2-layer GNN module.

GCN (Graph Convolutional Network)[16] is a semi-supervised learning approach for graph-structured data. It is based on an efficient convolutional neural network variation that works directly on graphs. A localized first-order approximation of spectral graph convolutions motivates the choice of convolutional architecture. The model learns hidden layer representations that encode both local graph structure and node attributes and scales linearly in the number of graph edges.

To give a mathematical formulation of GCN along with the previously defined norm, here we can use eq. 3 and eq. 4

$$m_v^{(l+1)} = \sum_{u\in\mathcal{N}(v)} \frac{e_{u,v} \cdot x_u^{(l)}}{|\mathcal{N}(v)|} \tag{3}$$

$$x_v^{(l+1)} = \sigma(W_l \cdot m_v^{(l+1)} + B_l \cdot x_v^{(l-1)}) \tag{4}$$

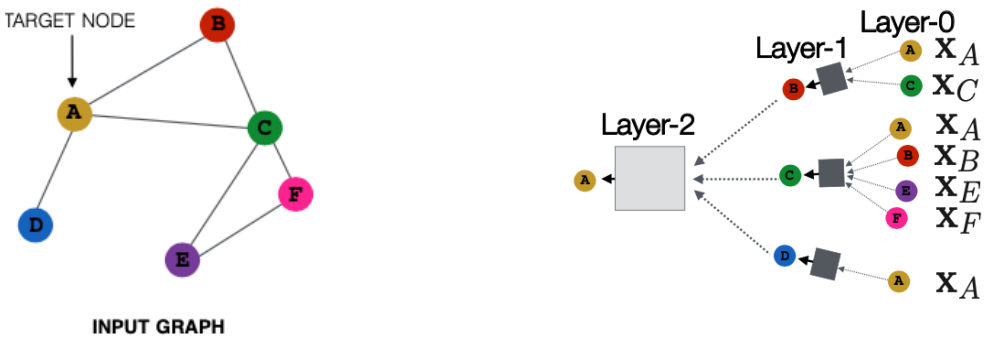

Figure 2: Abstract graph and corresponding computational graph

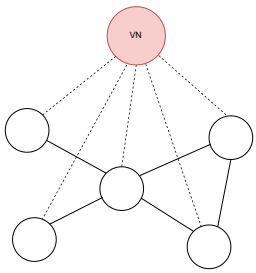

Figure 3: Illustration: Adding a virtual node in a 5-node graph

where e_u^v denotes the edge feature, $\sigma(\cdot)$ is a non-linear function, and W_l, B_l are trainable weight matrix parameters. GCN could be formalized using pure matrix operations as well (eq. 5):

$$\mathbf{X}^{(l+1)} = \sigma(\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}^{(l)}\mathbf{\Theta}) \tag{5}$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with processed self loops and $\hat{\mathbf{D}}$ is degree matrix of the graph, which makes $\hat{\mathbf{D}}_{i,i} = \sum_{j=0} \hat{\mathbf{A}}_{i,j}$, $\mathbf{X}^{(l)}$ is the representation of nodes in the graph, and $\mathbf{\Theta}$ denotes weight matrix.

GIN (Graph Isomorphism Network) [17] is proved to be able to significantly accurate to distinguish graph structures and is as powerful as the Weisfeiler-Lehman test. The node-wise formulation of GIN could written as eq. 6.

$$x_v^{(l+1)} = \mathrm{MLP}^{(l+1)}\left(\left(1+\epsilon^{(l+1)}\right)\cdot x_v^{(l)} + \sum_{u\in\mathcal{N}(v)} x_u^{(l)}\right) \tag{6}$$

GINE convolution operator [18], as shown in eq. 7, is a variant of GIN, which is able to corporate edge features into message passing framework, and that makes GIN have a higher performance.

$$x_v^{(l+1)} = \mathrm{MLP}^{(l+1)}\left(\left(1+\epsilon^{(l+1)}\right)\cdot x_v^{(l)} + \sum_{u\in\mathcal{N}(v)} \mathrm{ReLU}(x_u^{(l)} + e_{u,v})\right) \tag{7}$$

here $m_v^{(l+1)} = \sum_{u\in\mathcal{N}(v)} \mathrm{ReLU}(x_u^{(l)} + e_{u,v})$ and $\epsilon^{(l+1)}$ is a trainable scalar parameter.

Virtual Node Different from common approaches to extract the graph-level information using simple summing up or average[19], virtual node[12] provides a novel idea to profile the embedding of the graph, which is to append a virtual node to the given homogeneous graph that is connected to all other nodes, as shown in figure 3. The virtual node serves as a global scratch space that each node both reads from and writes to in every step of message passing. This allows information to travel long distances during the propagation phase. Node and edge features of the virtual node are added as zero-filled input features. Furthermore, special edge types will be added both for in-coming and out-going information to and from the virtual node.

2.3 Graph Neural Network Libraries

PyG (PyTorch Geometric) [10] is a library built upon PyTorch to easily compose and build up GNN learning pipelines for a wide range of applications related to structured data. It consists of various methods for deep learning on graphs and other irregular structures, also known as geometric deep learning, from a variety of published papers. In addition, it consists of easy-to-use mini-batch loaders for operating on many small and single giant graphs, multi GPU-support, DataPipe support, distributed graph learning via Quiver, a large number of common benchmark datasets (based on simple interfaces to create your own), the GraphGym experiment manager, and helpful transforms, both for learning on arbitrary graphs as well as on 3D meshes or point clouds.

DGL (Deep Graph Library)[11] is a Python package built for easy implementation of graph neural network model family, on top of existing DL frameworks (currently supporting PyTorch, MXNet and TensorFlow). It offers a versatile control of message passing, speed optimization via auto-batching and highly tuned sparse matrix kernels, and multi-GPU/CPU training to scale to graphs of hundreds of millions of nodes and edges[20].

3 Distributing Training and Systems

3.1 Distributed Training

Training deep learning models takes time. Deep neural networks often consist of millions or billions of parameters that are trained over huge datasets. As deep learning models become more complex, computation time can become unwieldy. Training a model on a single GPU can take weeks. Distributed training can split up the training workload and share among multiple hosts (computation nodes) that could work in parallel to fix this problem.

There are two main types of distributed training: data parallelism and model parallelism. In data parallelism, the dataset is divided into partitions, which will be distributed to every node later. The model is copied in each of the nodes, and is trained on the subset of the data it received. Each node independently computes the loss and gradients and then communicate with each other to keep their corresponding model parameters, or gradients synchronized at the end of the batch computation. In model parallelism, the model is segmented into several parts that can run sequentially in different nodes, and each one of them will run on the same data. Generally, it is more difficult to implement model parallelism than data parallelism.

In distributed training, speedup and efficiency are intuitive metrics measuring the relative performance improvement as the scale of computation resource enlarges. Mathematically speaking, speedup $S(x) = T(1)/T(x)$ and efficiency $E(x) = S(x)/x$, where $T(x)$ is time consumed using x unit(s) of computation resource.

3.2 Distributed Training using PyTorch

PyTorch Distributed Data-Parallel Training (DDP) is a widely adopted single-program multiple-data training paradigm. With DDP, the model is replicated on every process, and every model replica will be fed with a different set of input data samples. DDP takes care of gradient communication to keep model replicas synchronized and overlaps it with the gradient computations to speed up training. During the backwards pass, gradients from each node are averaged.

PyTorch Ignite [9] is a high-level library to help with training and evaluating neural networks written in PyTorch. Besides, it also provides high-level wrappers for PyTorch multiprocessing and DDP workflows.

4 Dataset

ogbg-molhiv is a public dataset offered by Open Graph Benchmark (OGB)[21], adapted from MoleculeNet. MoleculeNet[22] is a benchmark specially designed for testing machine learning methods of molecular properties. As the authors aim to facilitate the development of molecular machine learning method, this work curates a number of dataset collections, creates a suite of software that implements many known featurizations and previously proposed algorithms.

The HIV dataset was introduced by the Drug Therapeutics Program (DTP) AIDS Antiviral Screen, which tested the ability to inhibit HIV replication for over 40,000 compounds. Screening results were evaluated and placed into three categories: confirmed inactive (CI),confirmed active (CA) and confirmed moderately active (CM). The latter two labels are further combined , making it a classification task between inactive (CI) and active (CA and CM).

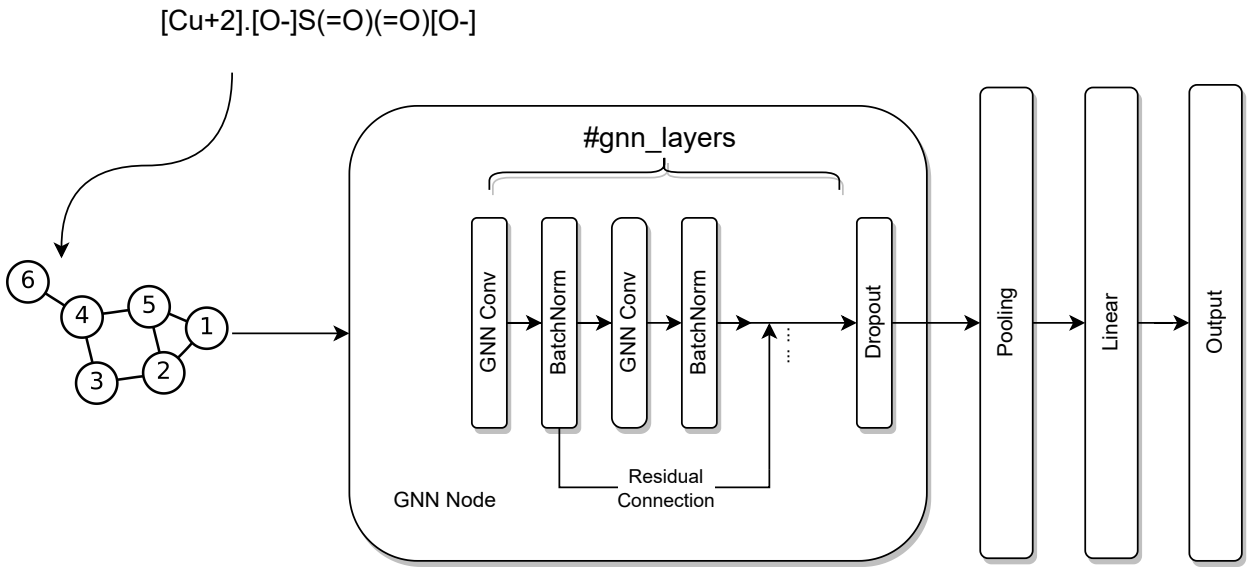

Figure 4: Architecture of designed model

5 Experiments

5.1 Model Design

Our model main body consists of 3 parts (figure 4):

- ***Data transformation and featurization***
  Simplified molecular-input line-entry system, known as "SMILES", is a specification in the form of a line notation for describing the structure of chemical species using short ASCII string, which was first initiated by David Weininger[23] in the 1980s. The data transformation and featurization of nodes and edges are processed with RDKit[24] to graph-structured objects.
- ***GNN convolutional layer stack***
  The GNN convolutional layer stack consists of sequential GNN convolutional layers, batch norm layers[25], dropout layers[26] and residual connections[27]. The whole stack serves as a way to iterate and abstract the embedding of the nodes in the graph for downstream tasks.
- ***Pooling layer and linear layer***
  Pooling layer profiles the graph using node embeddings and then pass the output to the linear layer, which outputs the finalized prediction of the task.

5.2 Experiment Setup and Hyperparameters

We primarily conduct experiments in the modes of: (I) PyG (II) DGL with PyTorch backend (III) PyG distributed mode (using Ignite) on a cluster with the scale of 1 (standalone and multiprocessing), 2, or 3.

To build up a distributed cluster, we use 3 computing instances of ‘lrz.medium‘ size on LRZ (Leibniz-Rechenzentrum), and each of them has 2 VCPUs, 9GB RAM, 20GB disk space with Ubuntu-18.04-LTS-bionic installed. The instances within the cluster communicated with each other via Munich Research Network (MWN). The hyperparameters in the learning process include: batch size=32, embedding dim=300, GNN layers=5, learning rate=0.001, dropout ratio=0.5, unless additionally specified.

5.3 Results

To investigate the performances in objective training modes, we implemented training modes mentioned above (Sec. 5.2) and plot the average time consumed per epoch (figure 5) in these modes and the evaluation of speedup and efficiency (figure 6) in parallel computing scenes. Training program implemented using PyG is generally faster than using DGL

	GCN	GCN VN	GIN	GIN VN
AUROC	0.767	0.764	0.787	0.769

Table 1: Model prediction performance

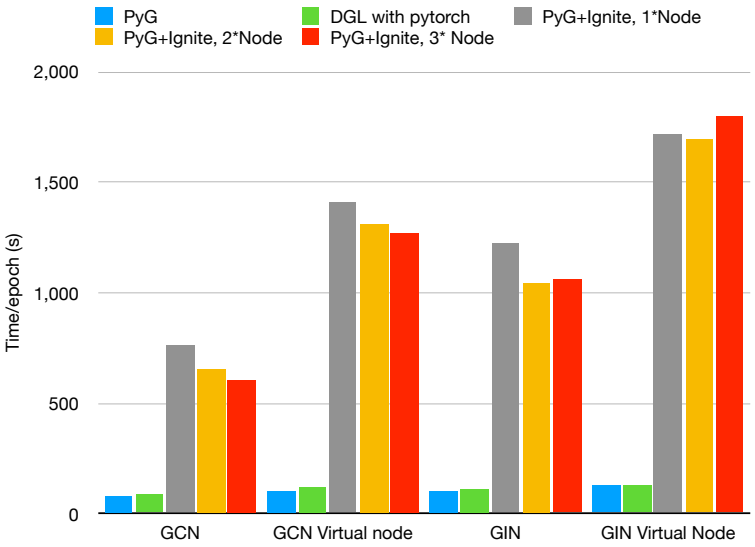

Figure 5: Training time in different training modes

with PyTorch backend. PyG implementation is 16%, 11%, and 4% faster than DGL one when using GCN, GCN virtual node, and GIN model respectively. Training a GCN model is always faster than training a GIN model, time is 34%, 18%, and 60% longer when training on PyG locally, DGL and PyG distributedly. Adding virtual node to GNN model increases the training time. Table 1 shows four models could achieve similar performance result with regard to AUROC metric on test data, but GIN outperforms than others.

It can also be obviously observed that using distributed training increases the training time significantly. In figure 5, the training time in a 3-node cluster using PyG+Ignite takes nearly 7.8 times of that using PyG locally, and in figure 6, the increase of speedup is minimal as we increase the number of nodes within the cluster, for example, the GCN model only has a speedup of $S(2)$ =1.15 and $S(3)$=1.25 which is against our intention of design. According to the discuss in PyTorch community[28], there are 3 time-consuming phases in a PyTorch distributed training loop, i.e. model forward, loss backward, and optimizer step. One possible and usual reason is that the lower rate of network communication slows down the overall training progress. Another possible reason is that frequently switching among the processes will be a heavy burden for the CPUs to work, and therefore slows down the training process.

To address this problem, we further record and analyze the time consumed in these phases, as table 2 shows. Comparing PyG, PyG+Ignite Local, PyG+Ignite Standalone Single processing, we can observe that using Ignite locally almost has no effect on training in all phases, however, PyG+Ignite Standalone Single processing mode slightly increases the training time and takes less time in forward but more time in loss backward. Comparing PyG+Ignite Standalone Single processing and PyG+Ignite Standalone Multiprocessing, it takes in all phases noticeably more time in multiprocessing mode, which explains process switching in multiprocessing mode consumes plenty of time. Comparing PyG+Ignite Standalone Multiprocessing and PyG+Ignite Distributed with 3 nodes, the rise in both absolute time from 201.84s to 387.15s and relative proportion of overall time from 34.51% to 64.49% of backward phase while the time of forward and optimizer step decreases could be explained by the communication delay caused by network when the data parallel strategy is adopted, since all communication in PyTorch DDP happens in backward phase and the rate of network becomes the bottleneck.

6 Conclusion

In this work, we implemented GNN (distributed) model training pipelines using PyG, DGL, and PyTorch Ignite, and evaluated the training performance among different libraries and distributing systems. Our conclusions include GIN is

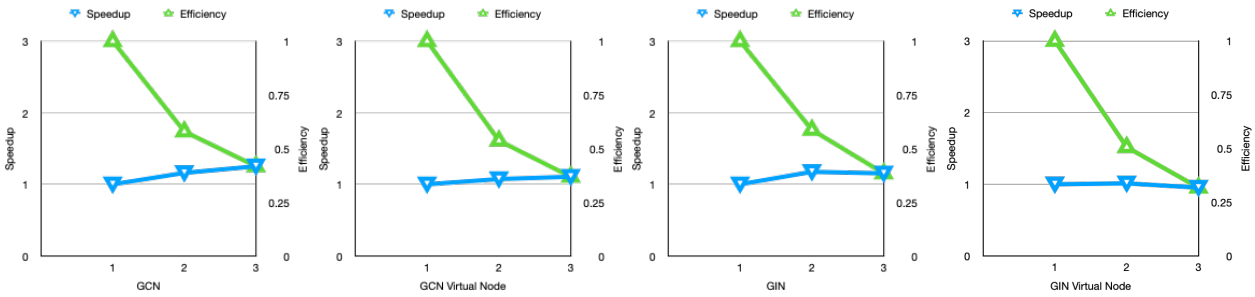

Figure 6: Speedup and efficiency plot in different training modes

	PyG		PyG+Ignite Local	
	Time (s)	Percent	Time (s)	Percent
Forward	42.14	54.71%	40.31	54.68%
Backward	29.69	38.55%	28.79	39.05%
Optim Step	5.18	6.72%	4.61	6.25%
Sum	77.01		73.72	

	PyG+Ignite Standalone Single processing		PyG+Ignite Standalone Multiprocessing		PyG+Ignite Distributed, 3 Nodes	
	Time (s)	Percent	Time/process (s)	Percent	Time/process (s)	Percent
Forward	24.89	30.01%	303.96	51.98%	183.37	30.54%
Backward	49.80	60.03%	201.84	34.51%	387.15	64.49%
Optim Step	8.25	9.94 %	78.92	13.49%	29.80	4.96%
Sum	82.95		584.72		600.32	

Table 2: Average time consumed of one iteration (batch size=32) in model forward, loss backward, and optimizer step phases in different settings.

more powerful to discriminate graphs, using PyG can make model training faster, network conditions have a crucial effect on distributed training system, and multiprocessing can slow down the training process.

Acknowledgments

This report is submitted in partial fulfillment of the assignment required by the Advantage Analytics and Machine Learning 2022SS course at Ludwig-Maximilians-Univeristät München.

References

[1] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems*, 30(11):3212–3232, 2019.

[2] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep learning in medical image analysis. *Annual review of biomedical engineering*, 19:221–248, 2017.

[3] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3):55–75, 2018.

[4] Hendrik Purwins, Bo Li, Tuomas Virtanen, Jan Schlüter, Shuo-Yiin Chang, and Tara Sainath. Deep learning for audio signal processing. *IEEE Journal of Selected Topics in Signal Processing*, 13(2):206–219, 2019.

[5] Aminu Da'u and Naomie Salim. Recommendation system based on deep learning methods: a systematic review and new directions. *Artificial Intelligence Review*, 53(4):2709–2748, 2020.

[6] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[7] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743, 2017.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[9] V. Fomin, J. Anmol, S. Desroziers, J. Kriss, and A. Tejani. High-level library to help with training neural networks in pytorch. `https://github.com/pytorch/ignite`, 2020.

[10] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[11] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[12] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[13] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015.

[14] Trang Pham, Truyen Tran, Hoa Dam, and Svetha Venkatesh. Graph classification via deep learning with virtual nodes. *arXiv preprint arXiv:1708.04357*, 2017.

[15] Jure Leskovec et al. Stanford cs224w.

[16] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[17] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[18] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. *arXiv preprint arXiv:1905.12265*, 2019.

[19] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. *Advances in neural information processing systems*, 28, 2015.

[20] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44. IEEE, 2020.

[21] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[22] Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. *Chemical science*, 9(2):513–530, 2018.

[23] David Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1):31–36, 1988.

[24] Greg Landrum et al. Rdkit: A software suite for cheminformatics, computational chemistry, and predictive modeling, 2013.

[25] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[28] Shen Li Jianfeng Wang. Distributed training is even slower sometimes. `https://discuss.pytorch.org/t/distributed-training-is-even-slower-sometimes/77927`, 2022.