A Practical Tutorial on Graph Neural Networks

ISAAC RONALD WARD, ISOLABS and the University of Southern California
JACK JOYNER and CASEY LICKFOLD, ISOLABS
YULAN GUO, Sun Yat-sen University
MOHAMMED BENNAMOUN, The University of Western Australia

Graph neural networks (GNNs) have recently grown in popularity in the field of artificial intelligence (AI) due to their unique ability to ingest relatively unstructured data types as input data. Although some elements of the GNN architecture are conceptually similar in operation to traditional neural networks (and neural network variants), other elements represent a departure from traditional deep learning techniques. This tutorial exposes the power and novelty of GNNs to AI practitioners by collating and presenting details regarding the motivations, concepts, mathematics, and applications of the most common and performant variants of GNNs. Importantly, we present this tutorial concisely, alongside practical examples, thus providing a practical and accessible tutorial on the topic of GNNs.

CCS Concepts: • **Computing methodologies** → **Neural networks**; *Artificial intelligence*;; • **Theory of computation** → **Graph algorithms analysis**

Additional Key Words and Phrases: Graph neural network, tutorial, artificial intelligence, recurrent, convolutional, auto encoder, decoder, machine learning, deep learning, papers with code, theory, applications

ACM Reference format:
Isaac Ronald Ward, Jack Joyner, Casey Lickfold, Yulan Guo, and Mohammed Bennamoun. 2022. A Practical Tutorial on Graph Neural Networks. *ACM Comput. Surv.* 54, 10s, Article 205 (September 2022), 35 pages.
https://doi.org/10.1145/3503043

I. R. Ward, J. Joyner, and C. Lickfold equal contribution.

This work was partially supported by ISOLABS, the Australian Research Council (Grants DP150100294 and DP150104251), the National Natural Science Foundation of China (No. U20A20185, 61972435), the Natural Science Foundation of Guangdong Province (2019A1515011271), and the Shenzhen Science and Technology Program (No. RCYX20200714114641140, JCYJ20190807152209394).

Authors' addresses: I. R. Ward, 311 Geneva St, Glendale, 91206, California, USA; email: isaacronaldward@gmail.com; J. Joyner, 35 Stirling Hwy, Crawley WA 6009, Australia; email: jack@isolabs.com.au; C. Lickfold, 14 Yilgarn Street, Shenton Park, 6008, Western Australia, Australia; email: casey@isolabs.com.au; Y. Guo, National University of Defense Technology 137 Yanwachi Changsha Hunan, 410073 People's Republic of China; email: yulan.guo@nudt.edu.cn; M. Bennamoun, 35 Stirling Hwy, Crawley WA 6009, Australia; email: mohammed.bennamoun@uwa.edu.au.

0360-0300/2022/09-ART205 $15.00
https://doi.org/10.1145/3503043

1 INTRODUCTION AND CONTEXT

Contemporary **artificial intelligence (AI)**, or more specifically, **deep learning (DL)**, has been dominated in recent years by the **neural network (NN)**. NN variants have been designed to increase performance in certain problem domains; the **convolutional neural network (CNN)** excels in the context of image-based tasks, and the **recurrent neural network (RNN)** in the space of **natural language processing (NLP)** and time series analysis. NNs have also been leveraged as building blocks in more complex DL frameworks—for example, they have been used as trainable generators and discriminators in **generative adversarial networks (GANs)** and as components in Transformer networks [86].

Graph neural networks (GNNs) provide a unified view of these input data types: The images used as inputs in computer vision, and the sentences used as inputs in NLP can both be interpreted as special cases of a single, general data structure—**the graph** (see Figure 1 for examples).

Formally, a graph is a set of distinct vertices (representing items or entities) that are joined optionally to each other by edges (representing relationships). Uniquely, the graphs fed into a GNN (during training and evaluation) **do not have strict structural requirements** per se; the number of vertices and edges between input graphs can change. In this way, GNNs can handle *unstructured*, *non-Euclidean* data [7], a property that makes them valuable in problem domains where graph data is abundant. Conversely, NN-based algorithms are typically required to operate on structured inputs with strictly defined dimensions. For example, a CNN built to classify over the MNIST dataset must have an input layer of 28×28 neurons, and all subsequent input images must be 28×28 pixels in size to conform to this strict dimensionality requirement [50].

The expressiveness of graphs as a method for encoding data and the flexibility of GNNs with respect to unstructured inputs has motivated their research and development. They represent a new approach for exploring relatively general DL methods, and they facilitate the application of DL approaches to sets of data that—until recently—were not not exposed to AI.

1.1 Contributions

The *key contributions* of this tutorial article are as follows:

(1) An easy-to-understand, introductory tutorial, which assumes no prior knowledge of GNNs.[1]
(2) Step-wise explanations of the mechanisms that underpin specific classes of GNNs, as enumerated in Table 1. These explanations progressively build a holistic understanding of GNNs.
(3) Descriptions of the advantages and disadvantages of GNNs and key areas of application.
(4) Full examples of how specific GNN variants can be applied to real-world problems.

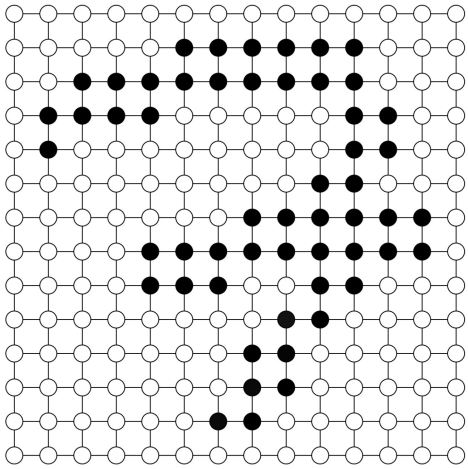
1.2 Taxonomy

The structure and taxonomy of this article is outlined in Table 1.

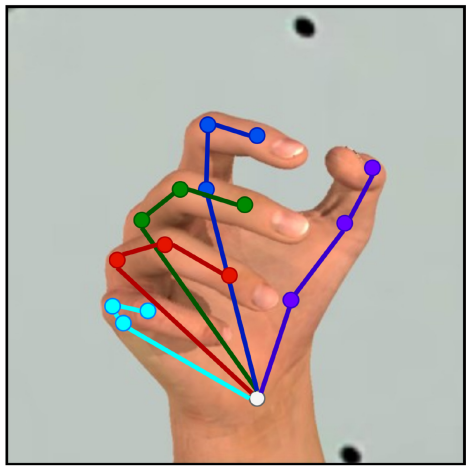
2 PRELIMINARIES

Here, we discuss some basic elements of graph theory, as well as the the key concepts required to understand how GNNs are formulated and operate. We present the notation that will be used consistently in this work (see Table 3).

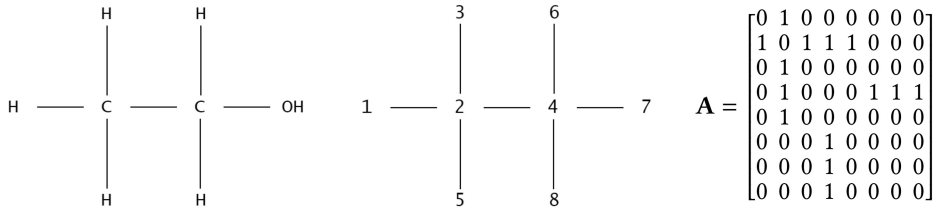
[1]We envisage that this work will serve as the "first port of call" for those looking to understand GNNs, rather than as a comprehensive survey of methods and applications. For those seeking a more comprehensive treatment, we highly recommend the following works: [30, 98, 108, 110] (see Table 2 for more details).



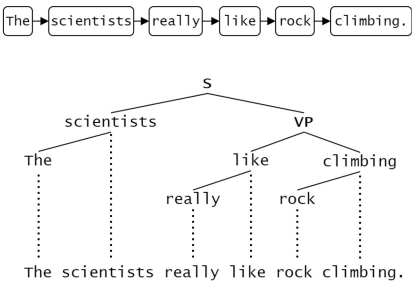
(a) A graph representation of a 14×14 pixel image of the digit '7'. Pixels are represented by vertices and their direct adjacency is represented by edge relationships.



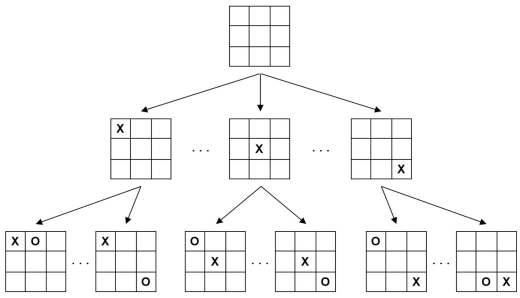
(b) A graph representing the joints in the human hand, and the hierarchical dependency of said joints. Images from the 'Hands from Synthetic Data' dataset [80].



(c) A diagram of an alcohol molecule (left), its associated graph representation with vertex indices labelled (middle), and its adjacency matrix (right).



(d) A vector representation and a Reed–Kellogg diagram (rendered according to modern tree conventions) of the same sentence. The graph structure encodes dependencies and constituencies.



(e) A gameplaying tree can be represented as a graph. Vertices are states of the game and directed edges represent actions which take us from one state to another.

Fig. 1. The graphs data structure is highly abstract and can be used to represent images (matrices), molecules, sentence structures, game playing trees, and so on.

Table 1. A Variety of Algorithms Are Discussed in This Tutorial Article

Broad class of algorithm	Related variants of algorithm
Recurrent GNNs (Section 3)	Graph LSTMs (Section 3.3), Gated GNNs (Section 3.3).
Convolutional GNNs (Section 4)	Spatial CGNNs (Section 4.2, including Graph Attention Networks, Message Passing Neural Networks, etc.), Spectral CGNNs (Section 4.3).
Graph Autoencoders (Section 5)	Variational Graph Autoencoders (Section 5.2), Graph Adversarial Techniques (Section 5.3).

This table illustrates potential use cases for each algorithm and the section where they are discussed. Should the reader prefer to read this tutorial article from an *applications/downstream task-based* perspective, then we invite them to review Tables 5, 6, and 8, which link each algorithm.

2.1 Key Terms

Graphs are formally defined by a set of vertices and the set of edges between these vertices: Put formally, $G = G(V, E)$. Fundamentally, graphs are just a way to encode data, and in that way, every property of a graph represents some real element, or concept in the data. Understanding how graphs can be used to represent complex concepts is key in appreciating their expressiveness and generality as an encoding device (see Figure 1 for examples of this domain agnostic expressiveness).

Vertices represent items, entities, or objects, which can naturally be described by quantifiable attributes and their relationships to other items, entities, or objects. We refer to a set of |V| vertices as V and the *i*th single vertex in the set as v_i. Note that there is no requirement for all vertices to be homogenous in their construction.

Edges represent and characterize the relationships that exist between items, entities, or objects. Formally, a single edge can be defined with respect to two (not necessarily unique) vertices. We refer to a set of |E| edges as E and a single edge between the *i*th and *j*th vertices as e_{ij}.

Neighborhoods are *subgraphs* within a graph and represent distinct groups of vertices and edges. Most commonly, the neighborhood $\mathcal{N}_{v_i}$ centered around a vertex v_i comprises of v_i, its adjoining edges (where $e_{ij} = 1$), and the vertices that are directly connected to it. Neighborhoods can be iteratively grown from a single vertex by considering the vertices attached (via edges) to the current neighborhood. Note that a neighborhood can be defined subject to certain vertex and edge feature criteria (i.e., all vertices within two hops of the central vertex, rather than one hop).

Features are quantifiable attributes that characterize a phenomenon that is under study. In the graph domain, features can be used to further characterize vertices and edges. Extending our social network example, we might have features for each person (vertex) that quantifies the person's age, popularity, and social media usage. Similarly, we might have a feature for each relationship (edge) that quantifies how well two people know each other, or the type of relationship they have (familial, colleague, etc.). In practice there might be many different features to consider for each vertex and edge, so they are represented by numeric feature vectors referred to as v_i^F and e_{ij}^F, respectively.

Embeddings are compressed feature representations. If we reduce large feature vectors associated with vertices and edges into low dimensional embeddings, then it becomes possible to classify them with low-order models (i.e., if we can make a dataset linearly separable). A key measure of an embedding's quality is if the points in the original space retain the same similarity in the embedding space. Embeddings can be created (or learned) for vertices, edges,

Table 2. A Comparison of Our Tutorial and Related Works

GNN papers	Main sections	Description
This work	Recurrent GNNs, Convolutional GNNs, Graph Autoencoders & Graph Adversarial Methods	A **tutorial paper** that steps through the operations of key GNN technologies in an explanatory and diagrammatic manner. Worked examples have been created to supplement explanations and are provided as code and in-text.
Graph Neural Networks: A Review of Methods and Applications [110]	GNN design framework, GNN modules, GNN variants, Theoretical and Empirical analyses & Applications	A **review paper** that proposes a general design framework for GNN models and systematically elucidates, compares, and discusses the varying GNN modules that can exist within the components of said framework.
Deep Learning on Graphs: A Survey [108]	Recurrent GNNs, Convolutional GNNs, Graph Autoencoders, Graph RL & Graph Adversarial Methods	A **survey paper** that outlines the development history and general operations of each major category of GNN. A complete survey of the GNN variants within said categories is provided (including links to implementations and discussions on computational complexity).
A Comprehensive Survey on Graph Neural Networks [98]	Recurrent GNNs, Convolutional GNNs, Graph Autoencoders & Spatial-temporal GNNs	A **survey paper** that provides a comprehensive categorization of contemporary GNN methods and benchmark datasets (across varying application domains). Numerous resources (e.g., open source code, datasets) are linked in a structured way.
Computing graph neural networks: A survey from algorithms to accelerators [1]	GNN fundamentals, modeling, applications, complexity, algorithms, accelerators & data flows	A **review** of the field of GNNs is presented from a computing perspective. A brief **tutorial** is included on GNN fundamentals, alongside an **in-depth** analysis of acceleration schemes, culminating in a communication-centric vision of GNN accelerators.

While other works provide comprehensive overviews of the field, our work focuses on explaining and illustrating key GNN techniques to the AI practitioner. Our goal is to act as a "first port of call" for readers, providing them with a basic understanding that they can build upon when reading more advanced material.

neighborhoods, or graphs. Embeddings are also referred to as representations, encodings, latent vectors, or high-level feature vectors, depending on the context.

Output types change depending on the problem domain. A GNN's forward pass can be thought of as two key processes: converting input graphs into useful embeddings, performing some downstream task (e.g., classification) on the embeddings, which converts the embeddings into some useful output. We define three commonly observed output types as follows:

Table 3. Notation Used in This Work

Notation	Meaning
V	A set of vertices.
|V|	The number of vertices in a set of vertices V.
v_i	The ith vertex in a set of vertices V.
v_i^F	The feature vector of vertex v_i.
E	A set of edges.
|E|	The number of edges in a set of edges E.
e_{ij}	The edge between the ith vertex and the jth vertex, in a set of edges E.
e_{ij}^F	The feature vector of edge e_{ij}.
$G = G(V, E)$	A graph defined by the set of vertices V and the set of edges E.
$\mathcal{N}_{v_i}$	The set of vertex *indices* for the vertices that are direct neighbors of v_i.
h_i^k	The kth hidden layer's representation of the ith vertex. Since each layer typically aggregates information from neighbors 1-hop away, this representation includes information from neighbors k-hops away.
o_i	The ith output of a GNN (indexing is dependant on output structure).
$\mathbf{I}_n$	An $n \times n$ identity matrix; all zero except for ones along the diagonal.
$\mathbf{A}$	The adjacency matrix; each element $\mathbf{A}_{ij}$ represents if the ith vertex is connected to the jth vertex by an edge.
$\mathbf{D}$	The degree matrix; a diagonal matrix of vertex degrees or valencies (the number of edges incident to a vertex). Formally defined as $\mathbf{D}_{i,i} = \sum_j \mathbf{A}_{ij}$.
$\mathbf{A}_W$	The weight matrix; each element $\mathbf{A}_{W\,ij}$ represents the "weight" of the edge between the ith vertex and the jth vertex. The "weight" typically represents some real concept or property. For example, the weight between two given vertices could be inversely proportional to their distance from one another (i.e., close vertices have a higher weight between them). Graphs with a weight matrix are referred to as *weighted graphs*, but not all graphs are weighted graphs; in unweighted graphs $\mathbf{A}_W = \mathbf{A}$.
$\mathbf{M}$	The incidence matrix; a |V| × |E| matrix where for each edge e_{ij}, the element of $\mathbf{M}$ at $(i, e_{ij}) = +1$, and at $(j, e_{ij}) = -1$. All other elements are set to zero. $\mathbf{M}$ describes the incidence of all edges to all vertices in a graph.
$\mathbf{L}$	The non-normalized combinatorial graph Laplacian; defined as $\mathbf{L} = \mathbf{D} - \mathbf{A}_W$.
$\mathbf{L}_{sn}$	The symmetric normalized graph Laplacian; defined as $\mathbf{L} = \mathbf{I}_n - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$.

We suggest that the reader familiarize themselves with this notation before proceeding.

(1) **Vertex-level** outputs require a prediction (e.g., a distinct class or regressed value) for each vertex in a given graph.
(2) **Edge-level** outputs require a prediction for each edge in a given graph.
(3) **Graph-level** outputs require a prediction per graph. For example: predicting the properties molecule graphs [96].

2.2 Learning Types

Transductive learning methods are exposed to all of the training and testing data before making predictions. For example: Our dataset might consist of a *single large graph* (e.g., Facebook's

social network graph) and the set of vertices is only partially labelled. The training set consists of the labelled vertices, and the testing set consists of both a small set of labelled vertices (for benchmarking) and the remaining unlabelled vertices. In this case, our learning methods should be exposed to the entire graph during training (including the test vertices), because the additional information (e.g., structural patterns) will be useful to learn from. Transductive learning methods are useful in such cases where it is challenging to separate the training and testing data without introducing biases.

Inductive learning methods reserve separate training and testing datasets. The learning process ingests the training data, and then the learned model is tested using the testing data, which it has not observed before in any capacity.

3 RECURRENT GRAPH NEURAL NETWORKS

In a standard NN, successive layers of learned weights work to extract progressively higher level features from an input tensor. In the case of NNs for computer vision, the presence of low-level features—such as short lines and curves—are identified by earlier layers, whereas the presence of high-level features—such as composite shapes—are identified by later layers. After being processed by these sequential layers, the resultant high-level features can then be provided to a softmax layer or single neuron for the purpose of classification, regression, or some other downstream task.

In the same way, the earliest GNNs extracted high-level feature representations from graphs by using successive feature extraction operations [60, 74] and then routed these high-level features to output functions. In other words: They processed inputs into useful embeddings and then processed embeddings into useful outputs using two distinct stages of processing. These early techniques had limitations: Some algorithms could only process **Directed Acyclic Graphs (DAGs)** [18], others required the input graphs to have "supersource" vertices (which had directed paths to all other vertices in the graph) [4], and some techniques required heuristic approaches to deal with the cyclical nature of certain graphs [61].

Typically, these early *recursive* methods relied on "unfolding" special cases of graphs into finite trees (recursive equivalents), which could then be processed into useful embeddings by recursive NNs [4]. The *Recurrent* GNN extended this, and thus provided a solution that could be applied to generic graphs [75]. Rather than create an embedding for the whole input graph via a recursive encoding network, RGNNs create embeddings at the vertex-level through an information propagation framework known as *message passing*, which will be defined in this section.

3.1 Recurrently Computing Embeddings

RGNNs compute embeddings at each vertex in the input graph using a deterministic, shared function called the *transition function*. It is named the transition function, as it can be interpreted as calculating the *next representation* of a neighborhood from the neighborhood's *current representation*. This transition function can be applied symmetrically at any vertex, even though the size of a vertex's neighborhood may be variable. This process is illustrated in Figure 2, where the transition function f calculates an embedding at each vertex for the surrounding neighborhood.

As such, the kth embedding h_i^k for any given vertex v_i is dependent on the following quantities:

- The features of the central vertex v_i^F.
- The features of all adjoining edges $e_{ij}^F, j \in \mathcal{N}_{v_i}$ (if edge features are present).
- The features of all neighboring vertices $v_j^F, j \in \mathcal{N}_{v_i}$.
- The previous iteration's embeddings of all neighboring vertices' $h_j^{k-1}, j \in \mathcal{N}_{v_i}$. $h_i^0 \ \forall i \in \mathrm{V}$ can be defined defined arbitrarily on initialization, and Banach's fixed point theorem will

guarantee that the subsequently calculated embeddings will converge to some optimal value exponentially (if f is implemented as a contraction map) [44].

To recurrently apply this learned transition function to compute successive embeddings, f must have a fixed number of input and output variables. How then can it be dependent on the immediate neighborhood, which might vary in size, depending on where we are in the graph? There are two simple solutions, the first of which is to set a "maximum neighborhood size" and use null vectors when dealing with vertices that have non-existing neighbors [74]. The second approach is to aggregate all neighborhood features in some permutation invariant manner [25], thus ensuring that any neighborhood in the graph is represented by a fixed size feature vector. While both approaches are viable, the first approach does not scale well to "scale-free graphs," which have degree distributions that follow a power law. Since many real world graphs (e.g., social networks) are scale-free [72], we will use the second solution here. Mathematically, this can be formulated as in Equation (1) [74].

$$h_i^k = \sum_{j \in \mathcal{N}_{v_i}} f\left(v_i^F, e_{ij}^F, v_j^F, h_j^{k-1}\right), \text{ where all } h_i^0 \text{ are defined on initialization.} \tag{1}$$

We can see that under this formulation Equation (1), f is well defined. It accepts four feature vectors that all have a defined length, regardless of which vertex in the graph is being considered, regardless of the iteration. This means that the transition function can be applied iteratively, until a stable embedding is reached for all vertices in the input graph. This expression can be interpreted as passing "messages," or features, throughout the graph; in every iteration, the embedding h_i^k is dependant on the features and embeddings of its neighbors. This means that with enough recurrent iterations, information will propagate throughout the whole graph: After the first iteration, any vertex's embedding encodes the features of the neighborhood within a range of a single edge.

In the second iteration, any vertex's embedding is an encoding of the features of the neighborhood within a range of two edges away, and so on. The iterative passing of "messages" to generate an encoding of the graph is what gives this *message passing* framework its name.[2]

Note that it is typical to explicitly add the identity matrix $\mathbf{I}_n$ to the adjacency matrix $\mathbf{A}$, thus ensuring that all vertices become trivially connected to themselves, meaning that a vertex $v_i \in \mathcal{N}_{v_i}$ $\forall i \in \mathrm{V}$. Moreover, this allows us to directly access the neighborhood by iterating through a single row of the adjacency matrix. This modified adjacency matrix is usually normalized to prevent unwanted scaling of embeddings.

3.2 Computing Downstream Outputs

Once we have useful embeddings centered around each vertex in the graph, the goal is to then inference meaningful outputs based on these values (i.e., to perform a *downstream* task). The **output** function g is responsible for taking the converged embeddings of a graph $G(V, E)$ and creating said output. In practice, the output function g, much like the transition function f, is implemented by a feed-forward neural network, though other means of returning a single value have been used, including mean operations, dummy super nodes, and attention sums [110].

Intuitively, the combined process of recurrently computing embeddings and subsequently computing downstream inputs can be interpreted as a sequential process of repeated NN computation blocks—or a finite computation graph (see Figure 2). In a supervised setting, a loss signal can be

[2]Importantly, this is not the formulation **Message Passing Neural Network (MPNN)** model [25], rather, it is a technique that uses the message passing framework. State-of-the-art approaches will be discussed in Section 4.2, and in particular, the MPNN will be defined explicitly.

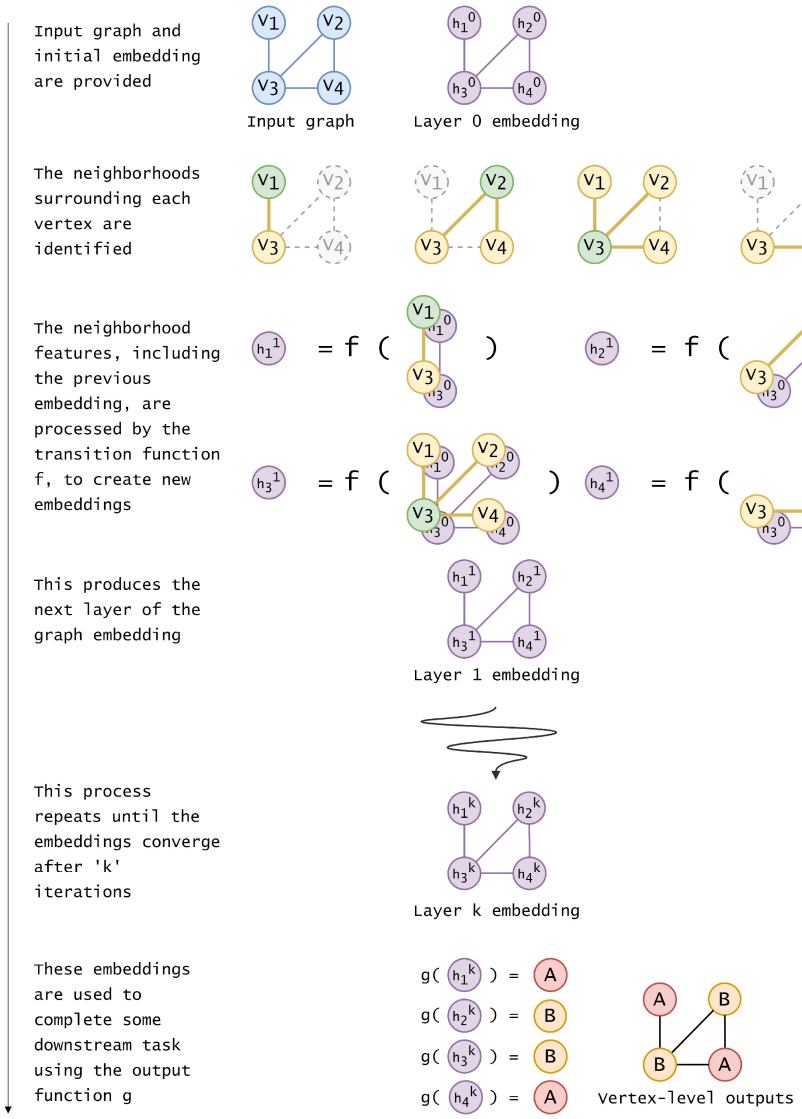


Fig. 2. An RGNN forward pass for a simple input graph $G(V, E)$ with $|V| = 4$, $|E| = 4$. G goes through k layers of processing. In each layer, each vertex's features v_i^F (green), the neighborhood's features $\mathcal{N}_{v_i}^F$ (yellow), and the previous hidden layer (purple) are processed by the *state transition function* f and aggregated, thereby producing successive embeddings of G. Note that the neighborhood features must be aggregated into a fixed embedding size, otherwise f would need to handle variable input sizes. This is repeated until the embeddings converge (i.e., the change between consecutive embeddings fails to exceed some stopping threshold). At that stage, the embeddings are fed to an *output function* g that perform some downstream task—in this case, the task is a vertex-level classification problem. Note that f and g can be implemented as NNs and trained via backpropagation of supervised error signals through the unrolled computation graph [60, 74]. Note that each vertex's embedding includes information from at max k "hops" away after the kth layer of processing. Image best viewed in color.

calculated that quantifies the error between the predicted output and a labelled ground truth. Both f and g can then be trained via backpropagation of errors, throughout the "unrolled" computation graph. For more detail on this process, see the calculations in [74].

3.3 Extensions for Sequential Graph Data

When discussing *recurrence* thus far, we have referred mainly to computing techniques that are iteratively applied to neighborhoods in a graph to produce embeddings that are dependent on information propagated throughout the graph. However, recurrent techniques may also refer to computing processes over *sequential data*, e.g., time series data. In the graph domain, sequential data refers to instances that can be interpreted as graphs with features that change over time. These include spatiotemporal graphs [98]. For example, Figure 1(b) illustrates how a graph can represent a skeletal structure in a single image of a hand, however, if we were to create such a graph for every frame of a contiguous video of a moving hand, then we would have a data structure that could be interpreted as a sequence of individual graphs, or a *single* graph with sequential features, and such data could be used for classifying hand actions in video.

As is the case with traditional sequential data, when processing each state of the sequence, we want to consider not only the current state but also information from the previous states, as outlined in Figure 6(a). A simple solution to this challenge might be to simply concatenate the graph emeddings of previous states to the features of the current state (as in Figure 6(b)), but such approaches do not capture long-term dependencies in the data. In this section, we outline how existing solutions from traditional DL—such as **Long Short-Term Memory Networks (LSTMs)** and **Gated Recurrent Units (GRUs)** (outlined in Figure 6)—can be extended to the graph domain.

Graph LSTMs (GLSTMs) make use of LSTM cells that have been adapted to operate on graph-based data. Whereas the aforementioned recurrent modules (Figure 6(b)) employ a simple concatenation strategy, GLSTMs ensure that long-term dependencies can be encoded in the LSTM's "cell state" (Figure 6(c)). This alleviates the vanishing gradient problem where long-term dependency signals are exponentially reduced when backpropagated throughout the network [35, 36].

GLSTM cells achieves this through **four** key processing elements that learn to calculate useful quantities based on the previous state's embedding and the input from the current state (as illustrated in Figure 6(c)).

(1) The **forget gate**, which uses L_f to extract values in the range [0,1], representing if elements in the previous cell's state should be "forgotten" (0) or retained (1).
(2) The **input gate**, which uses L_i to extract values in the range [0,1], indicating the amount of the modulated input that will be added to this cell's cell state.
(3) The **input modulation gate**, which uses L_n to extract values in the range [−1,1], representing learned information from this cell's input.
(4) The **output gate**, which uses L_o to calculate values in the range [0,1], indicating which parts of the cell state should be output as this cell's hidden state.

To use GLSTMs, we need to define all the operators in Figure 6(e). Since a graph $G(V, E)$ can be thought of as a variably sized set of vertices and edges, we can define graph concatenation as the separate concatenation of vertex features and edge features, where some null padding is used to ensure that the resultant tensor is of a fixed-size. This can be achieved by defining some "max number" of vertices for the input graphs. If the input signal for the GLSTM cell has a fixed size, then all other operators can be interpreted as traditional tensor operations, and the entire process is differentiable when it comes to backpropagation.

The Role of Recurrent Transitions in RGNNs for Graph Classification

In this independent example, we investigate social networks, which represent a rich source of graph data. Due to the popularity of social networking applications, accurate user and community classifications have become exceedingly important for the purpose of analysis, marketing, and influencing. In this example, we look at how the recurrent application of a transition function aids in making predictions on the graph domain, namely, in graph classification.

Dataset

We will be using the GitHub Stargazer's dataset [67] (available here). GitHub is a code sharing platform with social network elements. *Each* of the 12,725 graphs is defined by a group of users (vertices) and their mutual following relationships (undirected edges). Each graph is classified as either a web development group or a machine learning development group. There are **no vertex or edge features**—all predictions are made entirely from the structure of the graph.

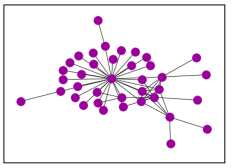


Fig. 3. A web developer group. |V| = 38 and |E| = 110.

Algorithms

Rather than use a true RGNN that applies a transition function to hidden states until some convergence criteria is reached, we will instead experiment with limited applications of the transition function. The transition function is a simple message passing aggregator that applies a learned set of weights to create size 16 hidden vector representations. We will see how the prediction task is affected by applying this transition function 1, 2, 4, and 8 times before feeding the hidden representations to an output function for graph classification. We train on 8,096 graphs for 16 epochs and test on 2,048 graphs for each architecture.

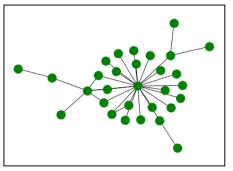


Fig. 4. A machine learning developer group. |V| = 30 and |E| = 66.

Results and Discussion

As expected, successive transition functions result in more discriminative features being calculated, thus resulting in a more discriminative final representation of the graph (analogous to more convolutional layers in a CNN).

Table 4. The effect of repeated transition function applications on graph classification performance

Algorithm	Acc. (%)	AUC
x1 transition	52%	0.5109
x2 transition	55%	0.5440
x4 transition	56%	0.5547
x8 transition	**64%**	**0.6377**

In fact, we can see that the final hidden representations become more linearly separable (see TSNE visualizations in Figure 5), thus, when they are fed to the output function—a linear classifier—the predicted classifications are more often correct. This is a difficult task, since there are no vertex or edge features. State-of-the-art approaches achieved the following mean AUC values averaged over 100 random train/test splits for the same dataset and task: GL2Vec [10]—0.551, Graph2Vec [62]—0.585, SF [16]—0.558, **FGSD [89]—0.656**.

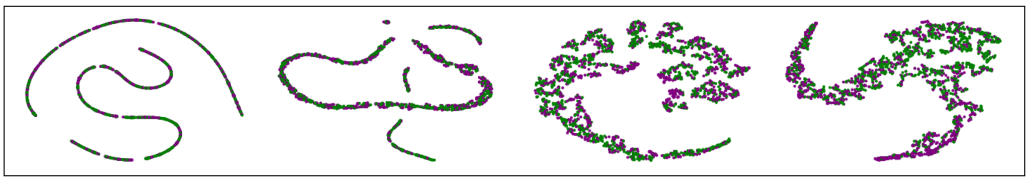


Fig. 5. TSNE renderings of final hidden graph representations for the x1, x2, x4, x8 hidden layer networks. Note that with more applications of the transition function (equivalent to more layers in a NN) the final hidden representations of the input graphs become *more* linearly separable into their classes (hence why they are able to be better classified using only a linear classifier).

Here, our transition function f was a "feedforward NN" with just one layer, so more advanced NNs (or other) implementations of f might result in more performant RGNNs. As more rounds of transition function were applied to our hidden states, the performance—and required computation—increased. Ensuring a consistent number of transition function applications is key in developing simplified GNN architectures and

in reducing the amount of computation required in the transition stage. We will explore how this improved concept is realized through CGNNs in Section 4.2.

Gated Recurrent Units (GRUs) provide a less computationally expensive alternative to GLSTMs by removing the need to calculate a cell state in each cell. As such, GRUs have **three** learnable weight matrices (as illustrated in Figure 6(d)) that serve similar functions to the four learnable weight matrices in GLSTMs. Again, GRUs require some definition of *graph concatenation*.

(1) The **reset gate** L_r determines how much information from to "forget" or "retain" when calculating the new information to add to the hidden state from the current state.
(2) The **update gate** L_u determines what information to "forget" or "retain" from the previous hidden state.
(3) The **candidate gate** L_n determines what information from the reset input will contribute to the next hidden state.

GRUs are well suited to sequential data when repeating patterns are less frequent, whereas LSTM cells perform well in cases where more frequent pattern information needs to be captured. LSTMs also have a tendency to overfit when compared to GRUs, and as such GRUs outperform LSTM cells when the sample size is low [27].

3.4 Advantages, Disadvantages, and Applications

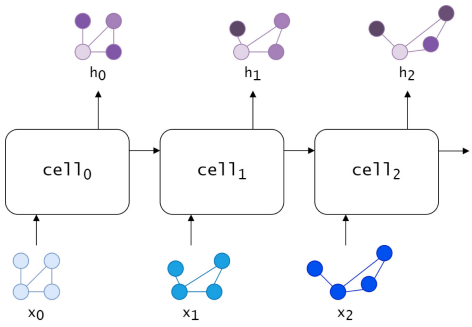
In this section, we have explained the forward pass that allows an RGNN to produce useful predictions over graph input data. During the forward pass, a transition function f is recursively applied to an input graph to create high-level features for each neighborhood. The repeated application of f ensures that at iteration k, an embedding h_i^k includes information from vertices k edges away from v_i. These high-level features can be fed to an output function to solve downstream tasks. During the backward pass, the parameters for the NNs f and g are updated with respect to a loss that is backpropagated through the computation graph defined in the forward pass. Recurrent processing units can also refer to approaches for handling graph-based sequential data, which include graph-based extensions to LSTMs and GRUs.

In actuality, the formulation for calculating embeddings provided in Equation (1) represents only one approach to calculating embeddings. This approach will be contextualised in Section 4, where a broader perspective on calculating useful embeddings will be introduced.

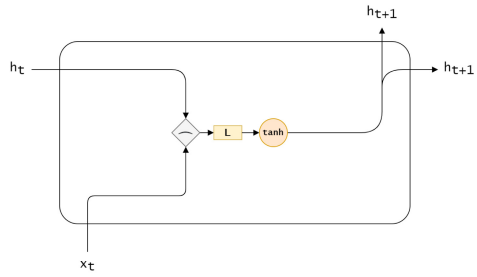
While RGNNs offer a simple approach to working with generic graphs, they have a number of shortcomings. Namely, the shared transition function f means that the same weights are being used to extract features in successive iterations, which may not be ideal for deep learning scenarios where the relationships between low-level features (earlier in the network) are different to the relationships between high-level features (later in the network). Moreover, since RGNNs iterate until convergence, they have variable length encoding networks, which can add implementation complexities. In the next section, we will discuss how these issues can be alleviated by developing formal definitions of convolution in the graph domain.

4 CONVOLUTIONAL GRAPH NEURAL NETWORKS

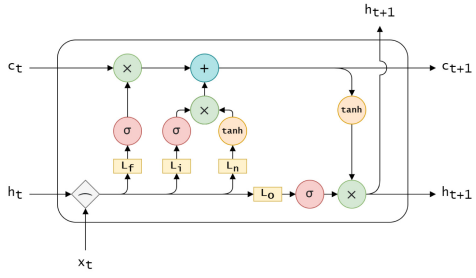
Convolutional NNs have achieved state-of-the-art performance on predictive tasks involving images. By convolving a learned kernel of weights with an input image, CNNs extract features of interest based on their visual appearance—regardless of their locality in the image. Since images are just a special case of graphs (see Figure 1(a)), a generalized convolution operator can be defined for the graph domain, thus bringing the following desirable properties to GNNs:



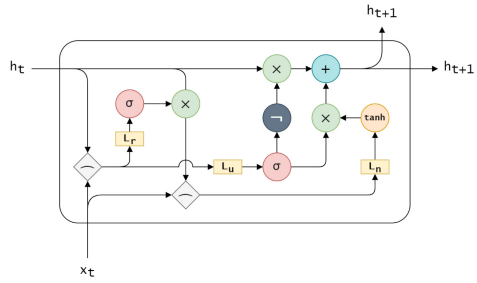
(a) The general processing approach for sequential graph data. The input data is a sequence of graphs (blue), and each processing cell considers not only the current input state, but also information from the states preceding it, thus yielding per-state embeddings (purple) which are dependent on the sequence thus far.



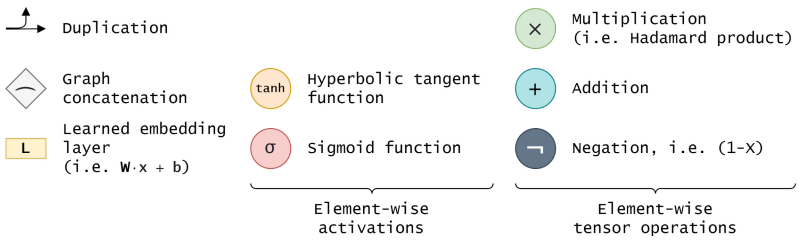
(b) A simple recurrent cell, which learns to extract useful features from the current input and the previous hidden state. After numerous cells of this type, the signal from the early input states is exponentially reduced.



(c) A GLSTM, which employs graph concatenation to enable LSTM-like processing of inputs. **Four** learned gates are employed to learn specific tasks within the cell.



(d) A Graph GRU, which employs graph concatenation to enable GRU-like processing of inputs. GRUs have relatively less learnable parameters than LSTMs, and are generally less prone to overfitting.

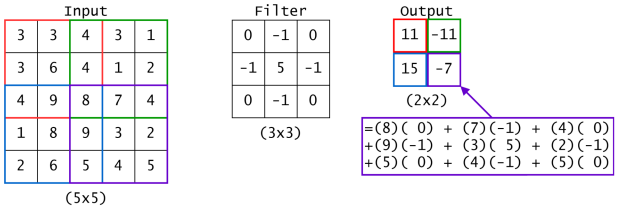


(e) Legend for the diagrams (a) – (d), all operators are traditional tensor operations apart from graph concatenation.

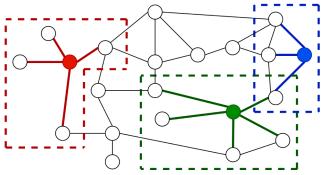
Fig. 6. The processing approaches for graph-based sequential data, including the overarching approach (a), simple RNN cells (b), GLSTMs (c), and graph GRUs (d).

Table 5. A Selection of Works That Use Recurrent GNN Techniques Such as Those Discussed in This Section

Approach	Applications
RNNs (early work) [61]	Quantitative structure-activity relationship analysis.
RNNs (early work) [4]	Various, including localization of objects in images.
RGNNs (early work) [74]	Various, including subgraph matching, the mutagenesis problem, and web page ranking.
RGNNs (Neural Networks for Graphs) [60]	Quantitative structure-activity relationship analysis of alkanes, and classification of general cyclic/acyclic graphs
RGNNs & RNNs (a comparison) [18]	4-class image classification
Geometric Deep Learning algorithms (incl. RGNNs) [7]	Graphs, grids, groups, geodesics, gauges, point clouds, meshes, and manifolds. Specific investigations include computer graphics, chemistry (e.g., drug design), protein biology, recommender systems, social networks, traffic forecasting, and so on.
RGNN pretraining [38]	Molecular property prediction, protein function prediction, binary graph classification, and so on.
RGNNs benchmarking [57]	Cycle detection and exploring what RGNNs can and cannot learn.
Natural Graph Networks (NGNs) [15]	Graph classification (bioinformatics and social networks).
GLSTMs [105]	Airport delay prediction (with $\|V\| = 325$).
GLSTMs (using differential entropy) [100]	Emotion classification from electroencephalogram (EEG) analysis (graphs calculated from K-nearest neighbor algorithms).
GLSTMs [58]	Speed prediction of road traffic throughout a directed road network (vertices are road segments, and edges are direct links between them).
GLSTMs (with spatiotemporal graph convolution) [63]	Real-time distracted driver behavior classification (i.e., based on the human pose graph [23] from a sequence of video frames—is the driver drinking, texting, performing normal driving, etc.). Other techniques for this problem include References [53, 79].
LSTM-Q (i.e., fusion of RL with a bidirectional LSTM) for graphs [13]	Connected autonomous vehicle network analysis for controlling agent movement (in a multi-lane road corridor).
Graph GRUs [54]	Computer program verification.
Graph GRUs [32]	Explainable predictive business process monitoring.
Graph GRUs [3]	NLP as a graph to sequence problem (leveraging structural information in language).
Graph GRUs [68, 69]	Gating for vertices and edges. Key applications include earthquake epicenter placement and synthetic regression problems.
Symmetric Graph GRUs [59]	Improved long-term dependency performance on synthetic tasks.



(a) A convolutional operation of 2D matrices. This process is used throughout computer vision and in CNNs. The convolutional operation here has a stride of 2 pixels. The given filter is applied in the red, green, blue, and then purple positions. At each position each element of the filter is multiplied with the corresponding element in the input (i.e., the Hadamard product) and the results are summed, producing a single element in the output. For clarity, this multiplication and summing process is illustrated for the purple position. In the case of this image the filter is a standard sharpening filter used in image analysis.



(b) Three neighborhoods in a given graph (designated by dotted boxes), with each one defined by a central vertex (designated by a correspondingly coloured circle). Each neighborhood in the graph is aggregated into a feature vector (i.e., and embedding) centered at each vertex, thus allowing the process to repeat for multiple layers.

Fig. 7. A comparison of image-based and graph-based spatial convolution techniques. Both techniques create embeddings centered around pixels/vertices, and the output of both techniques describes how the input is modified by the filter. Images best viewed in color.

(1) **Locality:** Learned feature extraction weights should be localized. They should only consider the information within a given neighborhood, and they should be applicable throughout the input graph.
(2) **Scalability:** The learning of these feature extractors should be scalable, i.e., the number of learnable parameters should be independent of |V|. Preferably the operator should be "stackable," so models can be built from successive independent layers, rather than requiring repeated iteration until convergence as with RGNNs in Section 3. Computation complexities should be bounded where possible.
(3) **Interpretability:** The convolutional operation should (preferably) be grounded in some mathematical or physical interpretation, and its mechanics should be intuitive to understand.

4.1 What Is Convolution?

We define convolution generally as an operation whereby **an output is derived from *two* given inputs by integration or summation, which expresses how the one is modified by the other**.

Convolution in CNNs involves two matrix inputs: one is the previous layer of activations, and the other is a matrix $W \times H$ of learned weights, which is "slid" across the activation matrix, aggregating each $W \times H$ region using a simple linear combination (see Figure 7(a)). In the spatial graph domain, it seems that this type of convolution is not well defined [78]; the convolution of a rigid matrix of learned weights must occur on a rigid structure of activation. How do we reconcile convolutions on unstructured inputs such as graphs?

Note that at no point during our general definition of convolution was the structure of the given inputs alluded to. In fact, convolutional operations can be applied to continuous functions (e.g., audio recordings and other signals), N-dimensional discrete tensors (e.g., semantic vectors in 1D, and images in 2D), and so on. During convolution, one input is typically interpreted as a filter (or *kernel*) being applied to the other input, and we will adopt this language throughout this

section. Specific filters can be utilized to perform specific tasks: In the case of audio recordings, high pass filters can be used to filter out low-frequency signals, and in the case of images, certain filters can be used to increase contrast, sharpen, or blur images. In our previous example of CNNs, filters are learned rather than designed.

4.2 Spatial Approaches

One might consider the early RGNNs described in Section 3 as using convolutional operations. In fact, these methods meet the criteria of locality, scalability, and interpretability. First, Equation (1) only operates over the neighborhood of the central vertex v_i, and can be applied on any neighborhood in the graph due to its invariance to permutation and neighborhood size. Second, the NN f is dependent on a fixed number of weights and has a fixed input and output that is independent of |V|. Finally, the convolution operation is immediately interpretable as a generalization of image-based convolution: In image-based convolution, neighboring pixel values are aggregated to produce embeddings; in graph-based spatial convolution, neighboring vertex features are aggregated to produce embeddings (see Figure 7). This type of graph convolution is referred to as the **spatial** graph convolutional operation, since spatial connectivity is used to retrieve the neighborhoods in this process.

Although the RGNN technique meets the definition of spatial convolution, there are numerous improvements in the literature. For example, the choice of aggregation function is not trivial—different aggregation functions can have notable effects on performance and computational cost.

A notable framework that investigated aggregator selection is the **GraphSAGE** framework [29], which demonstrated that learned aggregators can outperform simpler aggregation functions (such as taking the mean of embeddings) and thus can create more discriminative, powerful vertex embeddings. Regardless of the aggregation function, GraphSAGE works by computing embeddings based on the central vertex and an aggregation of its neighborhood (see Equation (2)). By including the central vertex, it ensures that vertices with near identical neighborhoods have different embeddings. GraphSAGE has since been outperformed on accepted benchmarks [20] by other frameworks [6], but the framework is still competitive and can be used to explore the concept of learned aggregators (see Section 4.3).

$$h_i^k = \sigma\left(\mathbf{W}\text{concat}\left(h_i^{k-1}, \text{aggregate}\left(h_j^{k-1}\ \forall j \in \mathcal{N}_{v_i}\right)\right)\right) \tag{2}$$

Alternatively, **Message Passing Neural Networks (MPNNs)** compute *directional* messages between vertices with a message function that is dependent on the source vertex, the destination vertex, and the edge connecting them [25]. Rather than aggregate the neighbor's features and concatenating them with the central vertex's features as in GraphSAGE, MPNNs sum the incoming messages, and pass the result to a readout function alongside the central vertex's features (see Equation (3)). Both the message function and readout function can be implemented with simple NNs in practice. This generalizes the concepts outlined in Equation (1) and allows for more meaningful patterns to be identified by the learned functions.

$$h_i^k = f_{readout}\left(h_i^{k-1}, \sum_{j \in \mathcal{N}_{v_i}} m_{ij}^k\right), \text{ where } m_{ij}^k = f_{message}\left(h_i^k, h_j^k, e_{ij}\right) \tag{3}$$

One of the most popular spatial convolution methods is **Graph Convolutional Networks (GCNs)**, which produce embeddings by summing features extracted from each neighboring

vertex and then applying non-linearity [97]. These methods are highly scalable, local, and furthermore, they can be "stacked" to produce layers in a CGNN. Each of these features is normalized based on the relative neighborhood scales of the current and neighbor vertex, thus ensuring that embeddings do not "explode" in scale during the forward pass.

$$h_i^k = \sigma\left(\sum_{j \in \mathcal{N}_{v_i}} \frac{\mathbf{W} h_j^{k-1}}{\sqrt{|\mathcal{N}_{v_i}||\mathcal{N}_{v_j}|}}\right) \tag{4}$$

Graph Attention Networks (GATs) extend GCNs: Instead of using the size of the neighborhoods to weight the importance of v_i to v_j, they implicitly calculate this weighting based on the normalized product of an attention mechanism [87]. In this case, the attention mechanism is dependent on the embeddings of two vertices and the edge between them. Vertices are constrained to only be able to attend to neighboring vertices, thus localizing the filters. GATs are stabilized during training using multi-head attention and regularization and are considered less general than MPNNs [88]. Although GATs limit the attention mechanism to the direct neighborhood, the scalability to large graphs is not guaranteed, as attention mechanisms have compute complexities that grow quadratically with the number of vertices being considered.

$$h_i^k = \sigma\left(\sum_{j \in \mathcal{N}_{v_i}} \alpha_{ij} \mathbf{W} h_j^{k-1}\right), \text{ where } \alpha_{ij} = \frac{e^{\text{att}(h_i^{k-1}, h_j^{k-1}, e_{ij})}}{\sum_{l \in \mathcal{N}_{v_i}} e^{\text{att}(h_i^{k-1}, h_l^{k-1}, e_{il})}} \tag{5}$$

Interestingly, all of these approaches consider information from the direct neighborhood and the previous embeddings, aggregate this information in some symmetric fashion, apply learned weights to calculate more complex features, and "activate" these results in some way to produce an embedding that captures non-linear relationships.

4.3 Spectral Approaches

In this section, we discuss another class of convolution approaches that evolved from the perspective of **Graph Signal Processing (GSP)** [78, 82]. These methods are attractive, as they are well grounded in a formal definition of convolution and can be directly interpreted as signal processing techniques in the domain of graph structured data.

The path to defining spectral graph convolution is described by the following series of statements.

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(u) g(t-u) du \tag{6}$$

(1) Defining a convolutional operator in the graph domain is desirable (as motivated in Section 4.2).
(2) From a signal processing perspective, the convolution operator is defined as in Equation (6). In other words, it is the integral of the product of a reversed and translated filter ($g(t-u)$) and an input function ($f(u)$). To define this in the graph domain, a translation operator needs to be defined for graphs.
(3) By Parseval's theorem, multiplication in the frequency domain (frequency space) corresponds to translation in the spatial domain (vertex space) [52]. Formally defining spatial translation in the graph domain requires a method to convert graphs between the vertex and frequency space.
(4) The eigenfunctions of the Laplacian define a basis in frequency space, so a formal definition of the graph Laplacian is required to develop spectral graph convolutions.

Using GraphSAGE to Generate Embeddings for Unseen Data

The **GraphSAGE** (**SA**mple and aggre**G**a**t**E) algorithm [29] emerged in 2017 as a method for not only learning useful vertex embeddings, but also for predicting vertex embeddings on *unseen* vertices. This allows powerful high-level feature vectors to be produced for vertices that were not seen at train time; enabling us to effectively work with dynamic graphs, or very large graphs (>100,000 vertices).

Dataset

In this example, we use the Cora dataset (see Figure 8) as provided by the deep learning library *DGL* [92]. The Cora dataset is oft considered "the MNIST of graph-based learning" and consists of 2,708 scientific publications (vertices), each classified into one of seven subfields in AI (or classes). Each vertex has a 1, 433 element binary feature vector, which indicates if each of the 1, 433 designated words appeared in the publication.

What is GraphSAGE?

GraphSAGE operates on a simple assumption: *vertices with similar neighborhoods should have similar embeddings*. In this way, when calculating a vertex's embedding, GraphSAGE considers the vertex's neighbors' embeddings. The function that produces the embedding from the neighbors' embeddings is learned, rather than the embedding being learned directly. Consequently, this method is **not** *transductive*, it is *inductive*, in that it generates general rules that can be applied to unseen vertices, rather than reasoning from specific training cases to specific test cases.

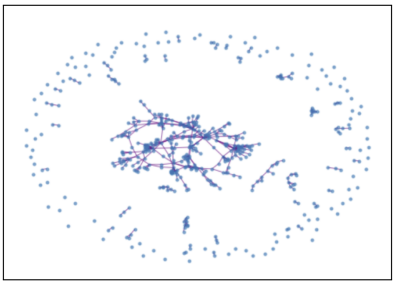


Fig. 8. A subgraph of the Cora dataset. The full Cora graph has |V| = 2708 and |E| = 5429. Note the many vertices with few incident edges (low degree) as compared to the few vertices with many incident edges (high degree).

Importantly, the GraphSAGE loss function is unsupervised and uses two distinct terms to ensure that neighboring vertices have similar embeddings and distant or disconnected vertices have embedding vectors that are numerically far apart. This ensures that the calculated vertex embeddings are highly discriminative.

Architectures

In this worked example, we experiment by changing the aggregator functions used in each GNN and observe how this affects our overall test accuracy. In all experiments, we use 2 hidden GraphSAGE convolution layers, 16 hidden channels (i.e., embedding vectors have 16 elements), and we train for 120 epochs before testing our vertex classification accuracy. We consider the **mean**, **pool**, and **LSTM (long short-term memory)** aggregator functions.

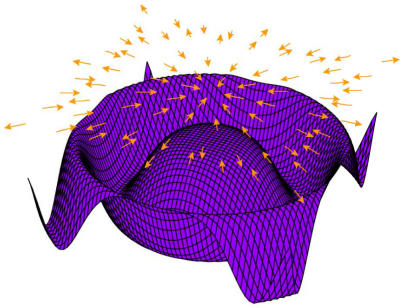
The **mean** aggregator function sums the neighborhood's vertex embeddings and then divides the result by the number of vertices considered. The **pool** aggregator function is actually a single fully connected layer with a non-linear activation function that then has its output element-wise max pooled. The layer weights are learned, thus allowing the most important features to be selected. The **LSTM** aggregator function is an LSTM cell. Since LSTMs consider input sequence order, this means that different orders of neighbor embedding produce different vertex embeddings. To minimize this effect, the order of the input embeddings is randomized. This introduces the idea of aggregator **symmetry**; an aggregator function should produce a constant result, invariant to the order of the input embeddings.

Results and Discussion

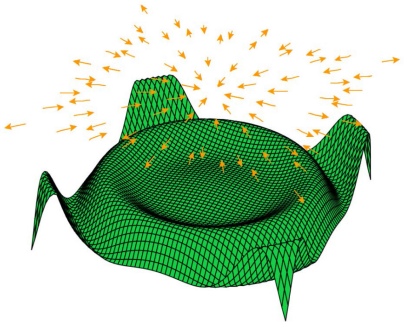
The mean, pool, and LSTM aggregators score test accuracies of **66.0**%, **74.4**%, and **68.3**%, respectively. As expected, the learned pool and LSTM aggregators are more effective than the simple mean operation, though they incur significant training overheads and may not be suitable for smaller training graphs or graph datasets. Indeed, in the original GraphSAGE paper [29], it was found that the LSTM and pool methods generally outperformed the mean and GCN aggregation methods across a range of datasets.

At the time of publication, GraphSAGE outperformed the state-of-the-art on a variety of graph-based tasks on common benchmark datasets. Since that time, a number of inductive learning variants of GraphSAGE have been developed, and their performance on benchmark datasets is regularly updated.[3]

[3]The state-of-the-art for vertex classification (Cora dataset): https://paperswithcode.com/sota/node-classification-on-cora.



(a) An input function $f(x, y)$ (rendered as a purple surface plot) and its gradient $\nabla f(x, y)$ (rendered as an orange vector field above the surface plot). The gradient, at every point, denotes the direction which increases $f(x, y)$ the most. In other words, the orange arrows always point in the direction of a maxima in the purple surface plot.



(b) The vector field from (a) $\nabla f(x, y)$ (rendered as an orange vector field above the surface plot) and its divergence $\nabla \cdot \nabla f(x, y)$ (rendered as a green surface plot). The divergence denotes how much every infinitesimal region of the vector field behaves like a source. In other words, it is a measure of the 'outgoing flow' of the infinitesimal volume at each point.

Fig. 9. An input function $f(x, y) : \mathbb{R}^2 \mapsto \mathbb{R}$ (a), its gradient $\nabla f(x, y) : \mathbb{R}^2 \mapsto \mathbb{R}^2$ ((a) and (b)), and the divergence of its gradient $\nabla \cdot \nabla f(x, y) : \mathbb{R}^2 \mapsto \mathbb{R}$ (b). The divergence of a function's gradient is known as the **Laplacian**, and it can be interpreted as measuring "how much" of a minimum each point is in the original function $f(x, y)$. The plots in (a) and (b) are an example of the entire calculation of the Laplacian; from scalar field to vector field (gradient), and then from vector field back to scalar field (divergence). The Laplacian is an analog of the second derivative and is often denoted by $\nabla \cdot \nabla$, ∇^2, or Δ.

The Laplacian is a second order differential operator that is calculated as the divergence of the gradient of a function in Euclidean space. The Laplacian occurs naturally in equations that model physical interactions, including but not limited to electromagnetism, heat diffusion, celestial mechanics, and pixel interactions in computer vision applications. Similarly, it arises naturally in the graph domain, where we are interested in the "diffusion of information" throughout a graph structure.

More formally, if we define flux as the quantity passing outward through a surface, then the Laplacian represents the density of the flux *of the gradient flow of a given function.* A step-by-step visualization of the Laplacian's calculation is provided in Figure 9. Note that the definition of the Laplacian is dependant on three things: **functions**, the **gradient** of a function, and the **divergence** of the gradient. Since we are seeking to define the Laplacian in the graph domain, we need to define how these constructs operate in the graph domain.

Functions in the graph domain (referred to as graph signals in GSP) are a mapping from every vertex in a graph to a scalar value: $f(G(V, E)) : \mathrm{V} \mapsto \mathbb{R}$. Multiple graph functions can be defined over a given graph, and we can interpret a single graph function as a single feature vector defined over the vertices in a graph. See Figure 10 for an example of a graph with two graph functions.

The **gradient of a function in the graph domain** describes the the direction and the rate of fastest increase of graph signals. In a graph structure, when we refer to "direction," we are referring to the edges of the graph; the avenues by which a graph function can change. For example, in Figure 10, the graph functions are 8-dimensional vectors (defined over the vertices), but the gradients of the functions for this graph are 12-dimensional vectors (defined over the edges) and are calculated as in Equations (7). Refer to Table 3 for a formal definition of the incident

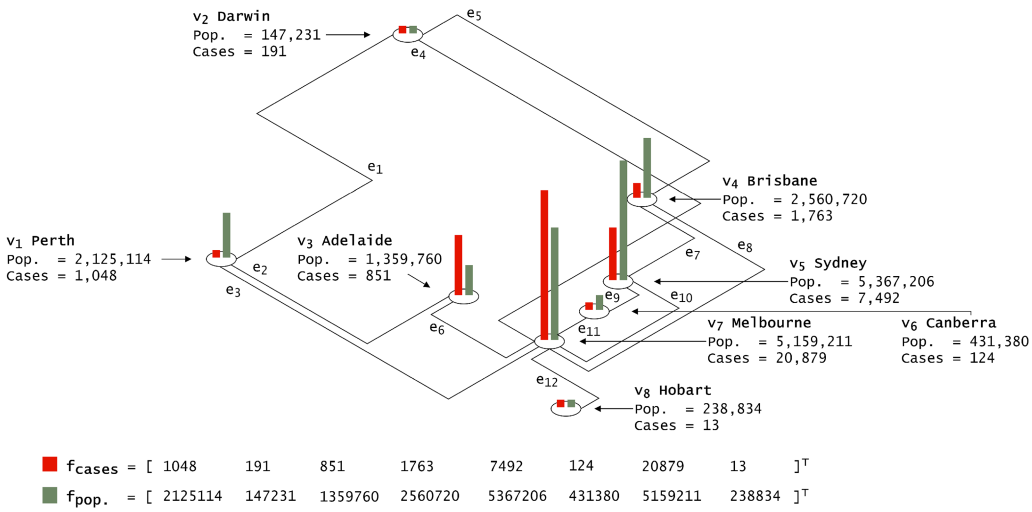


Fig. 10. A graph representing Australia (|V| = 8, |E| = 12). Its vertices represent Australia's capital cities, and the edges between them represent common flight paths. Each vertex has two features, one representing the population and another representing the total (statewide) cases of an infectious disease at those locations. Those two vertex feature vectors can be interpreted as the graph functions (also known as graph signals) f_{cases} and $f_{\text{pop.}}$, which are rendered at the bottom of the figure. As an example, it may be of interest to investigate the propagation/diffusion of these graph signal quantities throughout the graph structure.

matrix $\mathbf{M}$.

$$\mathbf{M}^{\mathrm{T}} = \begin{bmatrix} +1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ +1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ +1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & +1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & +1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & +1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & +1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & +1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & +1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & +1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & +1 & -1 \end{bmatrix}, \quad f_{\text{cases}} = \begin{bmatrix} 1,048 \\ 191 \\ 851 \\ 1,763 \\ 7,492 \\ 124 \\ 20,879 \\ 13 \end{bmatrix}, \quad \nabla f_{\text{cases}} = \mathbf{M}^{T} f_{\text{cases}} = \begin{bmatrix} +857 \\ \mathbf{+197} \\ -19,831 \\ -20,688 \\ -1,572 \\ -20,028 \\ -5,729 \\ -19,116 \\ +7,368 \\ -13,387 \\ -20,755 \\ \mathbf{+20,866} \end{bmatrix} \tag{7}$$

In Equations (7), the gradient vectors describe the difference in graph function value *across* the vertices/*along* the edges. Specifically, note that the largest magnitude value is 20,866 and corresponds to e_{12}, the edge between Hobart and Melbourne in Figure 10. In other words, the greatest magnitude edge is between the city with the least cases and the city with the most cases. Similarly, the lowest magnitude edge is e_2; the edge between Perth and Adelaide, which has the least difference in cases.

The **divergence of a gradient function in the graph domain** describes the outward flux of the gradient function at every vertex. To continue with our example, we could interpret the divergence of the gradient function f_{cases} as the outgoing "flow" of infectious disease cases from

each capital city. Whereas the gradient function was defined over the graph's edges, the divergence of a gradient function is defined over the graph's vertices and is calculated as in Equation (8).

$$\nabla \cdot (\nabla f_{\text{cases}}) = \mathbf{M}(\nabla f_{\text{cases}}) = \mathbf{M}(\mathbf{M}^T f_{\text{cases}}) = \mathbf{M}\mathbf{M}^T f_{\text{cases}} = \mathbf{L} f_{\text{cases}} = \begin{bmatrix} -18,777 \\ -23,117 \\ -20,225 \\ -23,273 \\ -290 \\ \mathbf{-28,123} \\ \mathbf{+134,671} \\ -20,866 \end{bmatrix} \tag{8}$$

The maximum value in the divergence vector for the infectious disease graph signal is 134,671, corresponding to Melbourne (the 7th vertex). Again, this can be interpreted as the magnitude of the "source" of infectious disease cases from Melbourne. Contrastively, the minimum value is −281,123, corresponding to Canberra, the largest "sink" of infections disease.

Note as well that the dimensionality of the original graph function is 8—corresponding to the vector space, its gradient's dimensionality is 12—corresponding to its edge space, and the Laplacian's dimensionality is again 8—corresponding to the vertex space. This mimics the calculation of the Laplacian in Figure 9, where the original scalar field (representing the magnitude at each point) is converted to a vector field (representing direction) and then back to a scalar field (representing how each point acts as a source).

The **graph Laplacian** appears naturally in these calculations as a $|\mathrm{V}| \times |\mathrm{V}|$ matrix operator in the form $\mathbf{L} = \mathbf{M}\mathbf{M}^T$ (see Equation (8)). This corresponds to the formulation provided in Table 3, as shown in Equation (9), and this formulation is referred to as the combinatorial definition $\mathbf{L} = \mathbf{D} - \mathbf{A}_W$ (the normalized definition is defined as $\mathbf{L}^{sn}$ [17]). The graph Laplacian is pervasive in the fields of GSP [14].

$$\mathbf{L} = \mathbf{M}\mathbf{M}^T = \begin{bmatrix} 3 & -1 & -1 & 0 & 0 & 0 & -1 & 0 \\ -1 & 3 & 0 & -1 & 0 & 0 & -1 & 0 \\ -1 & 0 & 2 & 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 3 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 3 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ -1 & -1 & -1 & -1 & -1 & -1 & 7 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} = \mathbf{D} - \mathbf{A}_W \tag{9}$$

Since $\mathbf{L} = \mathbf{D} - \mathbf{A}_W$, the graph Laplacian must be a real ($\mathbf{L}_{ij} \in \mathbb{R},\ \forall\ 0 \leq i, j < |\mathrm{V}|$) and symmetric ($\mathbf{L} = \mathbf{L}^T$) matrix. As such, it will have an **eigensystem** composed of a set of $|\mathrm{V}|$ orthonormal eigenvectors, each associated with a single real eigenvalue [78]. We denote the ith eigenvector with u_i, and the associated eigenvalue with λ_i, each satisfying $\mathbf{L}u_i = \lambda_i u_i$, where the eigenvectors u_i are the $|\mathrm{V}|$-dimensional columns in the matrix (Fourier basis) $\mathbf{U}$. The Laplacian can be factored as three matrices such that $\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$ through a process known as *eigenvector decomposition*. A variety of algorithms exist for solving this kind of eigendecomposition problem (e.g., the QR algorithm and Singular Value Decomposition).

These eigenvectors form a basis in $\mathbb{R}^{|\mathrm{V}|}$, and as such, we can express any discrete graph function as a linear combination these eigenvectors. We define the *graph Fourier transform* of any graph

function/signal as $\hat{f} = \mathbf{U}^T f \in \mathbb{R}^{|\mathrm{V}|}$, and its inverse as $f = \mathbf{U}\hat{f} \in \mathbb{R}^{|\mathrm{V}|}$. To complete our goal of performing convolution in the spectral domain, we now complete the following steps:

(1) Convert the lth graph function into the frequency space (i.e., generate its graph Fourier transform). We do this through matrix multiplication with the transpose of the Fourier basis: $\mathbf{U}^T f_l$. Note that multiplication with the eigenvector matrix is $O(N^2)$.
(2) Apply the corresponding lth learned filter in frequency space. If we define Θ_l as our lth learned filter (and a function of the eigenvalues of $\mathbf{L}$), then this appears like so: $\Theta_l \mathbf{U}^T f_l$.
(3) Convert the result back to vertex space by multiplying the result with the Fourier basis matrix. This completes the formulation defined in Equation (10). By Parseval's theorem, multiplication applied in the frequency space corresponds to translation in vertex space, so the filter has been *convolved* against the graph function [52].

$$\mathbf{U}\Theta_l \mathbf{U}^T f_l \tag{10}$$

This simple formulation has a number of downsides. Foremost, the approach is **not localized**—it has *global support*—meaning that the filter is applied to all vertices (i.e., the entirety of the graph function). This means that useful filters are not shared, and that the locality of graph structures is not being exploited. Second, it is **not scalable**: The number of learnable parameters grows with the size of the graph (not the scale of the filter) [17], the $O(N^2)$ cost of matrix multiplication scales poorly to large graphs, and the $O(N^3)$ time complexity of QR-based eigendecomposition [81] is prohibitive on large graphs. Moreover, directly computing this transform requires the diagonalization of the Laplacian and is *infeasible* for large graphs (where |V| exceeds a few thousand vertices) [31]. Finally, since the structure of the graph dictates the values of the Laplacian, **graphs with dynamic topologies cannot use this method of convolution**.

$$\Theta = \sum_{k=1}^{K} \theta_k T_k(\widetilde{\mathbf{\Lambda}}) \tag{11}$$

To alleviate the **locality** issue, Reference [21] noted that the smoothing of filters in the frequency space would result in localization in the vertex space. Instead of learning the filter directly, they formed the filter as a combination of smooth polynomial functions and instead learned the coefficients to these polynomials. Since the Laplacian is a local operator affecting only direct neighbors of any given vertex, then a polynomial of degree r affects vertices r-hops away. By approximating the spectral filter in this way (instead of directly learning it), spatial localization is thus guaranteed [77]. Furthermore, this improved **scalability**; learning K coefficients of the predefined smooth polynomial functions meant that the number of learnable parameters was no longer dependent on the size of the input graph. Additionally, the learned model could be applied to other graphs, too, as opposed to spectral filter coefficients that are basis-dependant. Since then, multiple potential polynomials have been used for specialized effects (e.g., Chebyshev polynomials, Cayley polynomials [51]).

Equation (11) outlines this approach. The learnable parameters are θ_k—vectors of Chebyshev polynomial coefficients—and $T_k(\widetilde{\mathbf{\Lambda}})$ is the Chevyshev polynomial of order k (dependent on the normalized diagonal matrix of scaled eigenvalues $\widetilde{\mathbf{\Lambda}}$). Chevyshev polynomials can be computed recursively with a stable recurrence relation and form an orthogonal basis [83]. We recommend Reference [65] for a full treatment on Chebyshev polynomials.

Interestingly, these approximate approaches demonstrate an equivalence between spatial and spectral techniques. Both are **spatially localized** and allow for a single filter to extract repeating patterns of interest throughout a graph, both have a number of learnable parameters that are

Table 6. A Particular Selection of Often-cited Works That Use Convolutional GNN Techniques (Such as Those Discussed in This Section)

Approach	Applications
GSP general (spectral) [82]	Multi-sensor temperature sensing (as a signal processing problem).
ChebNet (spectral) [83]	Various, but particularly in contexts where the functions to be approximated are high dimensional and smooth.
CayleyNets (spectral) [51]	Community detection, MNIST, citation networks, recommender systems, and other domains where specific frequency bands are of particular interest.
MPNNs (spatial) [25]	Quantum chemistry, specifically molecular property prediction.
GraphSAGE (spatial) [29]	Classifying academic papers, classifying Reddit posts, classifying protein functions, and so on.
GCNs (spatial) [45]	Semi-supervised vertex classification on citation networks and knowledge graphs.
Residual Gated Graph ConvNets (spatial) [6]	Subgraph matching and graph clustering.
Graph Isomorphism Networks (GINs) [99]	Various, including bioinformatics and social network datasets.
CGNN benchmarking [20]	Extensive, including ZINC [39], MNIST [49], CIFAR10 [48], and so on.
GATs [88]	Citation networks, protein-protein interaction.
GATs [73]	Robust pointwise correspondence of local image features.
Gated Attention Modules [106]	Traffic speed forecasting.
Edge GATs [11]	Citation networks, but generally any domain sensitive to relations/edge features.
Graph Attention Tracking [28]	Visual tracking (i.e., similarity matching between a template image and a search region).
Hyperbolic GATs [107]	Hyperbolic domains, e.g., protein surfaces, biomolecular interactions, drug discovery, or statistical mechanics.
Heterogeneous Attention Networks (HANs) [94]	Citation networks, IMBD (movie database networks), or any domain where vertices/edges are heterogeneous.
GATs [93]	Knowledge graphs and explainable recommender systems.
Graphormers [101]	Various, including quantum chemistry prediction. Particularly well suited to smaller scale graphs due to quadratic computation complexity of attention mechanisms.
Graph Transformers (with spectral attention) [47]	Various, including molecular graph analysis (i.e., Reference [39] and similar). Particularly well suited to smaller scale graphs as above.

Many of these algorithms are applicable to graph generally, and as such, the application column outlines the applications directly discussed in the cited paper.

independent of the input graph size, and each have meaningful and intuitive interpretations from a spatial (Figure 7) and spectral (Figure 10) perspective. In fact, GCNs can be viewed as a first-order approximation of Chebyshev polynomials [56]. For an in-depth treatment on the topic of GSP, we recommend References [82] and [78].

4.4 Advantages, Disadvantages, and Applications

As noted, spatial techniques and spectral techniques each have their advantages and disadvantages. The most popular spatial techniques are localized, generally scalable, and easily interpretable as methods for extracting features of interest from neighborhoods within graphs. As a downside, some of the more popular approaches (i.e., GATs) require expensive computations that scale in compute complexity quadratically with the size of their inputs, making them unsuitable for large graphs. However, spectral techniques can also be localized, scalable, and physically interpreted, but in some cases require rigorous computations to calculate the graph Laplacian. In general, eigendecomposition-based techniques cannot be used for graph inputs that have dynamic topologies and are computationally prohibitive for large graphs.

5 GRAPH AUTOENCODERS

GAEs represent the application of GNNs (often CGNNs) to autoencoding. The goal of an AE can be summarized as follows: to project the inputs features into a new space (known as the latent space) where the projection has more desirable properties than the input representation. These properties may include:

(1) The data being more separable (i.e., classifiable) in the latent space.
(2) The dimensionality of the dataset being smaller in the latent space than in the input space.
(3) The data being obfuscated for security or privacy concerns in the latent space.

A benefit of AEs in general is that they can often achieve this in an unsupervised manner—i.e., they can create useful embeddings without any training data. In their short history, GAEs have led the way in unsupervised learning on graph-structured data and enabled greater performance on supervised tasks such as vertex classification on citation networks [46].

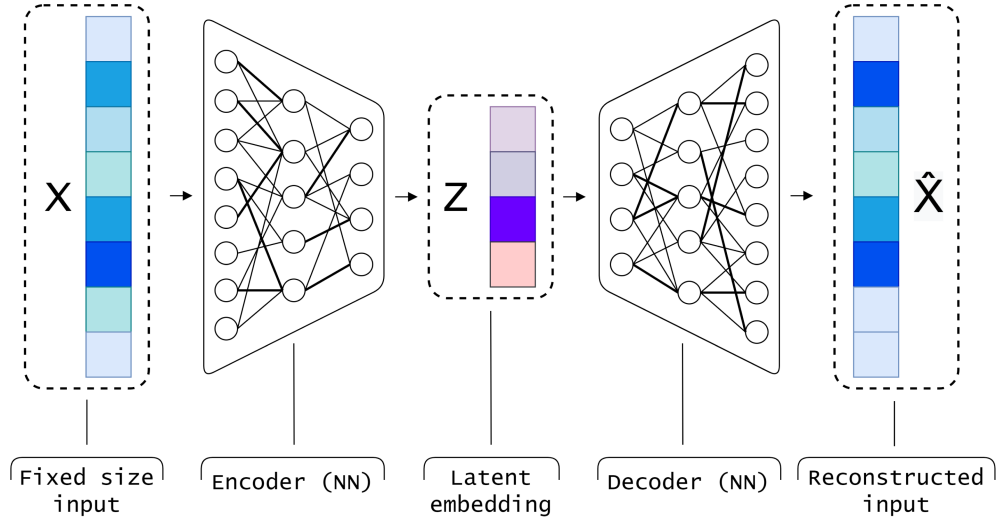
5.1 Autoencoders in the Graph Domain

AEs work in a two-step fashion, first encoding the input data into a latent space and then decoding this compressed representation to reconstruct the original input data, as depicted in Figure 11 (though in some cases higher dimensionality latent space representations have been used). The AE is then trained to minimize the *reconstruction loss*, which is calculated using only the input data and can therefore be trained in an unsupervised manner. In its simplest form, such a loss is defined as $\text{Loss}_{AE} = \|X - \hat{X}\|^2$, where we have an input instance X and the reconstructed input $\hat{X}$.

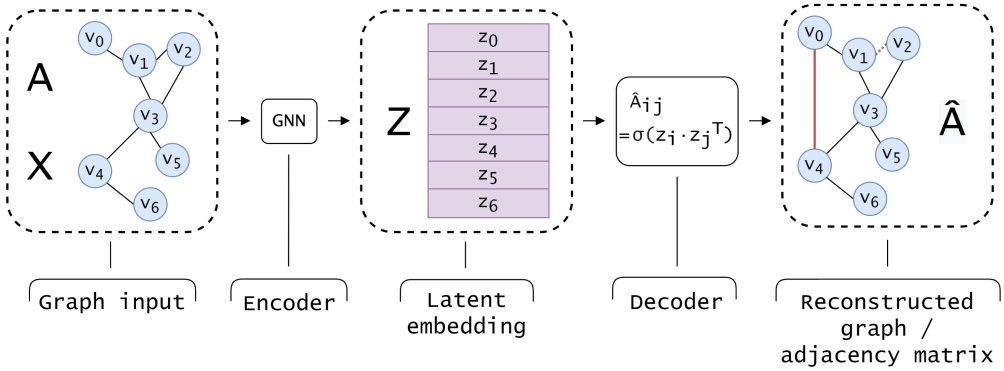
The difference between AEs to GAEs is illustrated in Figure 11 and requires the definition of encoders and decoders that take in and put out graph structures, respectively. One of the most common methods for doing this is to replace the encoder with a CGNN and replace the decoder with a method that can reconstruct the graph structure of the input [46].

With a well-defined loss function, we can perform end-to-end learning across this network to optimize the encoding and decoding to strike a balance between both **sensitivity** to inputs and **generalizability**—we do not want the network to overfit and "memorize" all training inputs. Rather, the goal is for the encoder network to represent repeating patterns within the input data in a more compressed and efficient format.

Once trained, GAEs (like AEs), can be split into their component networks to perform specific tasks. A popular use case for the encoder is to generating robust embeddings for supervised downstream tasks (e.g., classification, visualization, regression, clustering), and a use for the decoder is to generate new graph instances that include properties from the original dataset. This allows the generation of large synthetic datasets.



(a) The architecture for a simple traditional standard AE. AEs take a tensor input X, alter the dimensionality via a learnable encoder NN, and thus convert said input into a latent embedding Z. From there, the AE attempts to reconstruct the original input, thus creating the reconstructed input $\hat{X}$ (this process forms the decoder). By minimizing the reconstruction loss $\mathrm{L} = \|X - \hat{X}\|^2$, efficient latent space representations can be learned. This diagram shows an AE with a latent space representation that is smaller than the input size. In practice, encoder NNs can use custom layers and connections to improve performance.



(b) The architecture for a GAE. The input graph is described by the adjacency matrix **A** and the vertex feature matrix X in this case (though edge and global graph features can be accepted as input also). Since the input is an unstructured graph and not a tensor, a GNN architecture, such as those described throughout this tutorial, is used to generate a matrix of latent vertex embeddings Z. To reconstruct the input the similarity between all pairs of latent vertex embeddings is calculated, yielding a proxy for the 'connectedness' amongst the vertices in the graph. This creates the estimated adjacency matrix $\hat{A}$, which can be compared with the original **A** to create a loss term. In this example, the red edges denote edges which were incorrectly reconstructed.

Fig. 11. The architecture for a traditional tensor-based AE, compared to a GAE.

5.2 Variational Graph Autoencoders

Rather than representing inputs with single points in latent space, **variational autoencoders (VAEs)** learn to encode inputs as probability distributions in latent space. Figure 13 shows a VGAE that predicts a multivariate Guassian-like distribution $q(Z|A, X)$ for a given input.

Using Variational Graph Autoencoders for Unsupervised Learning

In this example, we will implement GAEs and VGAEs to perform unsupervised learning. After training, the learned embeddings can be used for both vertex classification and edge prediction tasks, even though the model was not trained to perform these tasks initially, thus demonstrating that the embeddings are meaningful representations of vertices. We will focus on edge prediction in citation networks, though these models can be easily applied to many task contexts and problem domains (e.g., vertex and graph classification).

Dataset

To investigate GAEs, we use the Citeseer, Cora, and PubMed datasets, which are accessible via PyTorch Geometric [24]. In each of these graphs, the vertex features are word vectors indicating the presence or absence of predefined keywords (see Section 4.3 for another example of the Cora dataset being used).

Algorithms

We first implement a GAE. This model uses a single GCN to encode input features into a latent space. An inner product decoder is then applied to reconstruct the input from the latent space embedding (as described in Section 5.2). During training, we optimize the network by reducing the reconstruction loss. We then apply the model to an edge prediction task to test whether the embeddings can be used in performing downstream machine learning tasks and not just in reconstructing the inputs.

We then implement the VGAE as first described in Reference [46]. Unlike GAEs, there are now two GCNs in the encoder model (one each for the mean and variance of a probability distribution). The loss is also changed to **Kullback–Leibler (KL)** divergence to optimize for an accurate probability distribution. From here, we follow the same steps as for the GAE method: An inner product decoder is applied to the embeddings to perform input reconstruction. Again, we will test the efficacy of these learned embeddings on downstream machine learning tasks.

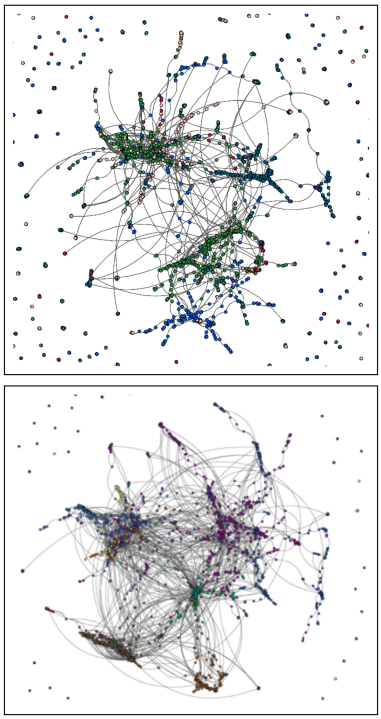


Fig. 12. Renderings of the Citeseer dataset (top) and Cora dataset (bottom). Image best viewed in colour.

Results and Discussion

To test GAEs and VGAEs on each graph, we average the results from 10 experiments. In each experiment, the model is trained for 200 iterations to learn a 16-dimensional embedding.

Table 7. Comparing the link prediction performance of autoencoder models on Citeseer.

Algorithm	Dataset	AUC	AP[a]
GAE	Citeseer	0.858 (±0.016)	0.868 (±0.009)
VGAE	Citeseer	**0.869 (±0.007)**	**0.878 (±0.006)**
GAE	Cora	0.871 (±0.018)	0.890 (±0.013)
VGAE	Cora	**0.873 (±0.01)**	**0.892 (±0.008)**
GAE	PubMed	**0.969 (±0.002)**	**0.971 (±0.002)**
VGAE	PubMed	0.967 (±0.003)	0.696 (±0.003)

In alignment with Reference [46], we see the VGAE outperform GAE on the Cora and Citeseer graphs, while the GAE outperforms the VGAE on the PubMed graph. Performance of both algorithms was significantly higher on the PubMed graph, likely owing to PubMed's larger number of vertices (|V| = 19,717), and therefore more training examples, than Citeseer (|V| = 3,327) or Cora (|V| = 2,708). Moreover, while Cora and Citeseer vertex features are simple binary word vectors (of sizes 1,433 and 3,703, respectively), PubMed uses the more descriptive TF-IDF word vector, which accounts for the frequency of terms. This feature may be more discriminative and thus more useful when learning vertex embeddings.

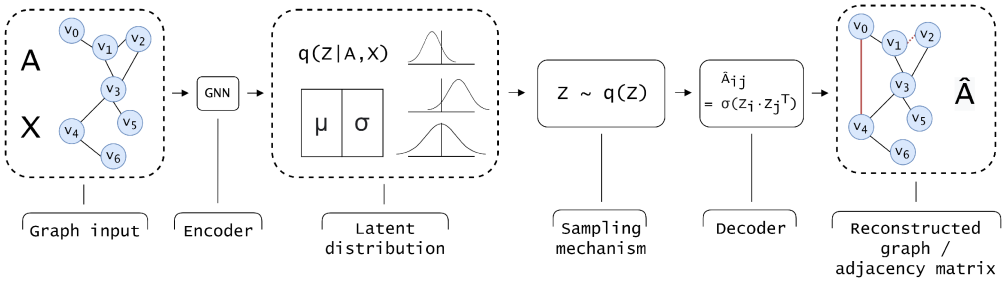


Fig. 13. An example of a VGAE. Graph inputs are encoded via a GNN into multivariate Guassian parameters (i.e., mean and variance). These represent ranges of possible values in the latent space, which enforces a continuous latent space representation. Samples are selected from these distributions and fed to the decoder, as in Figure 11(b). In practice, researchers have observed that this method ensures that all regions of latent space map to meaningful outputs and that latent vectors that are close to one another map to reconstructions that are "close to" one another in the input space. To ensure that the encoded distributions are well behaved, a penalty term is added to the loss function to enforce the distributions to match some known prior distribution (i.e., normal distributions). The total loss function for a VGAE is thus defined as $\mathrm{L} = \|X - \hat{X}\|^2 + KL((N(0,1), q(Z))$.

This creates a "smoother" latent space that covers the full spectrum of inputs, rather than leaving "gaps," where an unseen latent space vector would be decoded into a meaningless output. This has the effect of increasing generalization to unseen inputs and regularizing the model to avoid overfitting. Ultimately, this approach transforms the GAE into a more suitable generative model.

Unlike in GAEs—where the loss is simple the mean squared error between the input and the reconstructed input—a VGAE's loss imposes an additional penalty that ensures that the latent distributions are normalized. More specifically, this term regularizes the latent space distributions by ensuring that they do not diverge significantly from some prior distribution with desirable properties. In our example, we use the normal distribution (denoted as $N(0,1)$). This divergence is quantified in our case using **Kulback-Leibler divergence (KL)**, though other similarity metrics (e.g., Wassertein space distance or ranking loss) can be used successfully. Without this loss penalty, the VGAE encoder might generate distributions with small variances or high magnitude means; both of which would make it harder to sample from the distribution effectively.

5.3 Improving Robustness with Graph Adversarial Techniques

Graph Adversarial Techniques (GAdvTs) use adversarial learning methods, whereby an AI model acts as an adversary to another during training to mutually improve the performance of both models in tandem. Due to the adversarial nature of GAdvT's, developments in this area have been described as an "arms race between attackers and defenders" [12]. As with traditional adversarial techniques, common goals for GAdvTs include:

- Improving the robustness, regularization, or distribution of learned embeddings.
- Improving the robustness of models to targeted attacks.
- Training generative AI models.

The field of GAdvTs is broad, with multiple different kinds of attacks, training regimes, and use cases. In this tutorial, we will look at how GAdvTs can be used to extend VGAEs to create robust encoding networks, and well regularized generative mechanisms. Figure 14 describes a typical architecture for adversarially training a VGAE.

To ensure that the sampling operation is differentiable, VGAEs leverage a "reparameterization trick," where a random Guassian sample is generated from $N(0,1)$ *outside* the forward pass of the

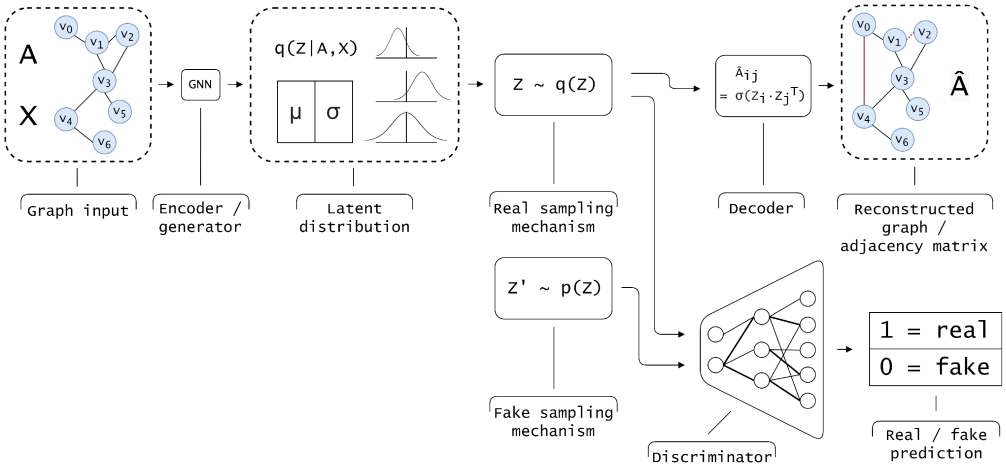


Fig. 14. A typical approach to adversarial training with VGAEs. The top row described a VGAE as illustrated in Figure 13. Importantly, for each real sample, a "fake" sample is generated from some prior distribution $p(Z)$ (e.g., a multivariate Gaussian or some other distribution that is believed to model the properties of the latent space attributes). During training, these fake and real samples are input into a discriminator network, which predicts whether said inputs are real or fake. If the discriminator *correctly* classifies the sample, then the generator is *penalized*, thus optimizing the encoder to generate distributions whose samples are more likely to "fool" the discriminator. In other words, this causes the encoder to create samples that have similar properties to the samples pulled from the prior distribution $p(z)$, thus acting as a form of regularization.

network. The sample is then transformed by the parameterization of the generated distribution $q(z)$, rather than having the sample be generated directly from $q(z)$ [19]. Since this approach is entirely differentiable, it allows for end-to-end training via backpropagation of an unsupervised loss signal.

5.4 Advantages, Disadvantages, and Applications

In this section, we have explained the mechanics behind traditional and graph autoencoders. Analogous to how AEs use NN to perform encoding, GAEs use CGNNs to perform encoding and create embeddings [46]. Similarly, an unsupervised reconstruction error term is defined; in the case of GAEs, this is between the original adjacency matrix and the predicted adjacency matrix (produced by the decoder). GAEs and VGAEs represent a simple method for performing unsupervised training, which allows us to learn powerful embeddings in the graph domain without any labelled data, but requires regularization techniques to smooth their latent space representations and reparameterization tricks to ensure differentiability.

Before the seminal work in Reference [46] on VGAEs, a number of deep GAEs had been developed for unsupervised training on graph-structured data, including **Deep Neural Graph Representations (DNGR)** [8] and **Structure Deep Network Embeddings (SDNE)** [91]. These methods operate on only the adjacency matrix, so information about both the entire graph and the local neighborhoods is lost. More recent work mitigates this by using an encoder that aggregates information from a vertex's local neighborhood to learn latent vector representations. For example, Reference [70] proposes a linear encoder that uses a single weights matrix to aggregate information from each vertex's one-step local neighborhood, showing competitive performance on numerous benchmarks. Despite this, typical GAEs use more complex encoders—primarily CGNNs—to capture nonlinear relationships in the input data and larger local neighborhoods [5, 8, 46, 64, 84, 85, 91, 102].

Table 8. A Selection of Works Using GAE/VGAE/GAdvT Techniques as Discussed in This Section

Approach	Applications
Deep Neural Graph Representation [8]	Various, including clustering, calculating useful vertex embeddings, and visualization.
Structure Deep Network Embeddings [91]	Various, including language networks, citation networks, and social networks.
Denoising Attribute AEs [34]	Various, including social networks and citation networks.
Link prediction-based GAEs (and VGAEs) [71]	Various, including link prediction and bidirectionally prediction on citation networks.
VGAEs [46]	Various, including citation networks.
Deep Gaussian Embedding of Graphs (G2G) [5]	Various, including citation networks.
Semi-implicit VGAEs [33]	Various graph analytic tasks, including citation networks.
Adversarially Regularized AEs (NetRA) [102]	Directed communication networks, software class dependency networks, undirected social networks, citation networks, directed word networks with inferred "Part-of-Speech" tags, and Protein-Protein Interactions.
Adversarially Regularized Graph Autoencoder (ARGA, and its variants) [64]	Various, including vertex clustering and visualization of citation networks.
Graph Convolutional Generative Adversarial Networks [90]	Traffic prediction in optical networks (particularly in domains with "burst events").
FeederGAN (adversarial) [55]	Generation of distributed feeder circuits.
Labelled Graph GANs [22]	Generating graph-structured data with vertex labels. Demonstrated for citation networks and protein graphs.
Graph GANs for Sparse Data [43]	Generating sparse graph datasets. Demonstrated for MNIST and high-energy physics proton-proton jet particle data.
Graph Convolutional Adversarial Networks [37]	Predicting missing infant diffusion MRI data for longitudinal studies.

6 FUTURE RESEARCH

The field of GNNs is rapidly developing, and there are numerous directions for meaningful future research. In this section, we outline a few specific directions that have been identified as important research areas to focus on [98, 103, 108, 110].

6.1 Explainability

Recent advancements in deep learning have allowed deeper NNs to be developed and applied throughout the field of AI. As the mechanics that drive predictions (and thus decisions) become more complex, the path by which those decisions are reached becomes more obfuscated. **Explainable AI (XAI)** promises to address this issue.

Explainability in the graph domain promises much of the same benefits as it does across AI, including more interpretable outputs, clearer relationships between inputs and outputs, more interpretable models, and in general, more trust between AI and human operators across problem domains (e.g., digital pathology [40], knowledge graphs [95]).

While the suite of available XAI algorithms has been consistently growing over the recent years (e.g., LIME, SHAP), graph specific XAI algorithms are relatively few and far between [103]. A key reason for this might be the requirement for graph explanations to incorporate not just the relationships among the input features, but also the relationships surrounding the input's structural/-topological information. In particular, the exploration of instance-level explainers—including high-fidelity perturbative and gradient-based methods—may provide good approximations of input importance in graph prediction tasks. Further techniques, especially those that assign quantitative importances to a graph's structure *and* its features will give a more holistic view of explainability in the graph domain.

6.2 Scalability

In traditional deep learning, a common technique for dealing with extremely large datasets and AI models is to distribute and parallelize computations where possible. In the graph domain, the non-rigid structure of graphs presents additional challenges. For example, how can a graph be uniformly partitioned across multiple devices? How can message-passing frameworks be efficiently implemented in a distributed system? These questions are especially pertinent for extremely large graphs. Recent developments suggest that sampling-based approaches may provide appropriate solutions in the near future [109], though such solutions are non-trivial, especially when graphs are stored on distributed systems [76].

Moreover, the scalability of GNN modules themselves may be improved by further directed research. For example, popular GNN variants such as MPNNs can in practice only be applied to small graphs due to the large computational overheads associated with the message-passing framework. Methods such as GATs show promising results regarding scalability, but attentional mechanisms still incur a quadratic time complexity, which may be prohibitive for graphs with large neighborhoods (on average). An exciting further avenue of research regarding GATs is their equivalence to Transformer networks [41, 47, 86, 101]. Further directed research in this area may contribute not only to the development of exciting new graph-based techniques, but also the understanding of Transformer networks as a whole. Breakthroughs in this area may address challenges specific to Transformers, such as the design of efficient positional encodings, effective warm-up strategies, and the quantification of inductive biases.

6.3 Advanced Learning Paradigms

Self-supervised Learning (SSL) techniques have recently been suggested as the "next step" in AI training paradigms, as they close the gap—and even outperform—fully supervised approaches in visual tasks [2, 9]. Contemporary approaches to SSL include using models that learn from one another [9, 26, 104]. In related work, recent research suggests that contrastive objectives can be designed in the graph domain by selecting views of a single graph instance, thus permitting the capture of universal structural properties across graph without the need for large labelled datasets [42, 66, 111, 112]. These initial investigations demonstrate that the graph domain is well suited to the application of advanced learning paradigms techniques. Further research in this area may produce more general pretrained GNNs, allow the leveraging of large unlabelled graph datasets, and yield further insight into nature unsupervised/weakly supervised learning in the development of intelligence.

7 CONCLUSION

The development of GNNs has accelerated *hugely* in the recent years due to increased interest in exploring unstructured data and developing general AI solutions. In this article, we have illustrated key GNN variants, described the mechanisms that underpin their operations, addressed their limitations, and worked through examples of their application to various real-world problems (with links to more advanced literature where necessary). Going forward, we expect that GNNs will continue to emerge as an exciting and highly performant branch of algorithms that natively model and address important real-world problems.

ACKNOWLEDGMENTS

We express a special appreciation to Josh Crowe at ISOLABS for his ongoing support of technical research (including this tutorial article) at ISOLABS. We also thank Richard Pienaar for providing early feedback that greatly improved this work.

REFERENCES

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2021. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Comput. Surv.* 54, 9 (2021), 1–38.

[2] Mahmoud Assran, Mathilde Caron, Ishan Misra, Piotr Bojanowski, Armand Joulin, Nicolas Ballas, and Michael Rabbat. 2021. Semi-supervised learning of visual features by non-parametrically predicting view assignments with support samples. *arXiv preprint arXiv:2104.13963*.

[3] Daniel Beck, Gholamreza Haffari, and Trevor Cohn. 2018. Graph-to-sequence learning using gated graph neural networks. *arXiv preprint arXiv:1806.09835*.

[4] M. Bianchini, M. Gori, and F. Scarselli. 2002. Recursive processing of cyclic graphs. In *Proceedings of the International Joint Conference on Neural Networks*. IEEE, 154–159.

[5] Aleksandar Bojchevski and Stephan Günnemann. 2018. Deep Gaussian embedding of graphs: Unsupervised inductive learning via ranking. *arXiv: Machine Learning*.

[6] Xavier Bresson and Thomas Laurent. 2017. Residual Gated Graph ConvNets. CoRR abs/1711.07553. http://arxiv.org/abs/1711.07553.

[7] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. 2017. Geometric deep learning: Going beyond Euclidean data. *IEEE Sig. Process. Mag.* 34, 4 (2017), 18–42.

[8] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2016. Deep neural networks for learning graph representations. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI'16)*. AAAI Press, 1145–1152.

[9] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. 2021. Emerging properties in self-supervised vision transformers. *arXiv preprint arXiv:2104.14294*.

[10] Hong Chen and Hisashi Koga. 2019. GL2vec: Graph embedding enriched by line graphs with edge features. In *Proceedings of the International Conference on Neural Information Processing*.

[11] Jun Chen and Haopeng Chen. 2021. Edge-featured graph attention network. *arXiv preprint arXiv:2101.07671*.

[12] Liang Chen, Jintang Li, Jiaying Peng, Tao Xie, Zengxu Cao, Kun Xu, Xiangnan He, and Zibin Zheng. 2020. A survey of adversarial learning on graphs. *arXiv preprint arXiv:2003.05730*.

[13] Sikai Chen, Jiqian Dong, Paul Ha, Yujie Li, and Samuel Labi. 2021. Graph neural network and reinforcement learning for multi-agent cooperative control of connected autonomous vehicles. *Comput.-Aid. Civil Infrastr. Eng.* 36, 7 (2021), 838–857.

[14] Fan R. K. Chung and Fan Chung Graham. 1997. *Spectral Graph Theory*. Number 92. American Mathematical Society.

[15] Pim de Haan, Taco Cohen, and Max Welling. 2020. Natural Graph Networks. arXiv:cs.LG/2007.08349.

[16] Nathan de Lara and Edouard Pineau. 2018. A simple baseline algorithm for graph classification. *CoRR* abs/1810.09155 (2018). arXiv:1810.09155 http://arxiv.org/abs/1810.09155.

[17] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. *Adv. Neural Inf. Process. Syst.* 29 (2016), 3844–3852.

[18] Vincenzo Di Massa, Gabriele Monfardini, Lorenzo Sarti, Franco Scarselli, Marco Maggini, and Marco Gori. 2006. A comparison between recursive neural networks and graph neural networks. In *Proceedings of the IEEE International Joint Conference on Neural Network Proceedings*. IEEE, 778–785.

[19] Carl Doersch. 2016. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*.

[20] Vijay Prakash Dwivedi, Chaitanya K. Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2020. Benchmarking Graph Neural Networks. arXiv:cs.LG/2003.00982.

[21] Joan Bruna Estrach, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2014. Spectral networks and deep locally connected networks on graphs. In *Proceedings of the 2nd International Conference on Learning Representations*.
[22] Shuangfei Fan and Bert Huang. 2019. Labeled graph generative adversarial networks. *arXiv preprint arXiv:1906.03220*.
[23] Hao-Shu Fang, Shuqin Xie, Yu-Wing Tai, and Cewu Lu. 2017. RMPE: Regional multi-person pose estimation. In *Proceedings of the International Conference on Computer Vision*.
[24] Matthias Fey and Jan E. Lenssen. 2019. Fast graph representation learning with PyTorch geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.
[25] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural message passing for quantum chemistry. In *Proceedings of the International Conference on Machine Learning*. PMLR, 1263–1272.
[26] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, et al. 2020. Bootstrap your own latent: A new approach to self-supervised learning. *arXiv preprint arXiv:2006.07733*.
[27] Nicole Gruber and Alfred Jockisch. 2020. Are GRU cells more specific and LSTM cells more sensitive in motive classification of text? *Front. Artif. Intell.* 3 (2020), 40.
[28] Dongyan Guo, Yanyan Shao, Ying Cui, Zhenhua Wang, Liyan Zhang, and Chunhua Shen. 2021. Graph attention tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9543–9552.
[29] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *CoRR* abs/1706.02216.
[30] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation learning on graphs: Methods and applications. *CoRR* abs/1709.05584.
[31] David K. Hammond, Pierre Vandergheynst, and Rémi Gribonval. 2011. Wavelets on graphs via spectral graph theory. *Appl. Comput. Harmon. Anal.* 30, 2 (2011), 129–150.
[32] Maximilian Harl, Sven Weinzierl, Mathias Stierle, and Martin Matzner. 2020. Explainable predictive business process monitoring using gated graph neural networks. *J. Decis. Syst.* (2020), 1–16.
[33] Arman Hasanzadeh, Ehsan Hajiramezanali, Nick Duffield, Krishna R. Narayanan, Mingyuan Zhou, and Xiaoning Qian. 2019. Semi-Implicit Graph Variational Auto-Encoders. arXiv:cs.LG/1908.07078.
[34] Bhagya Hettige, Weiqing Wang, Yuan-Fang Li, and Wray Buntine. 2020. Robust attribute and structure preserving graph embedding. In *Advances in Knowledge Discovery and Data Mining*, Hady W. Lauw, Raymond Chi-Wing Wong, Alexandros Ntoulas, Ee-Peng Lim, See-Kiong Ng, and Sinno Jialin Pan (Eds.). Springer International Publishing, Cham, 593–606.
[35] Sepp Hochreiter. 1991. Untersuchungen zu dynamischen neuronalen Netzen. *Diploma, Technische Universität München* 91, 1 (1991).
[36] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. 2001. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press In.
[37] Yoonmi Hong, Jaeil Kim, Geng Chen, Weili Lin, Pew-Thian Yap, and Dinggang Shen. 2019. Longitudinal prediction of infant diffusion MRI data via graph convolutional adversarial networks. *IEEE Trans. Med. Imag.* 38, 12 (2019), 2717–2725.
[38] Weihua Hu,* Bowen Liu,* Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. 2020. Strategies for pre-training graph neural networks. In *Proceedings of the International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=HJlWWJSFDH.
[39] John J. Irwin, Teague Sterling, Michael M. Mysinger, Erin S. Bolstad, and Ryan G. Coleman. 2012. ZINC: A free tool to discover chemistry for biology. *J. Chem. Inf. Model.* 52, 7 (2012), 1757–1768. DOI: https://doi.org/10.1021/ci3001277
[40] Guillaume Jaume, Pushpak Pati, Antonio Foncubierta-Rodriguez, Florinda Feroce, Giosue Scognamiglio, Anna Maria Anniciello, Jean-Philippe Thiran, Orcun Goksel, and Maria Gabrani. 2020. Towards explainable graph representations in digital pathology. *arXiv preprint arXiv:2007.00311*.
[41] Chaitanya Joshi. 2020. Transformers are graph neural networks. *The Gradient* (2020). Retrieved from https://thegradient.pub/transformers-are-gaph-neural-networks/.
[42] Nikola Jovanović, Zhao Meng, Lukas Faber, and Roger Wattenhofer. 2021. Towards robust graph contrastive learning. *arXiv preprint arXiv:2102.13085*.
[43] Raghav Kansal, Javier Duarte, Breno Orzari, Thiago Tomei, Maurizio Pierini, Mary Touranakou, Jean-Roch Vlimant, and Dimitrios Gunopulos. 2020. Graph generative adversarial networks for sparse data generation in high energy physics. *arXiv preprint arXiv:2012.00173*.
[44] M. A. Khamsi and William A. Kirk. 2001. *An Introduction to Metric Spaces and Fixed Point Theory*. Wiley.
[45] Thomas N. Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *CoRR* abs/1609.02907.
[46] Thomas N. Kipf and Max Welling. 2016. Variational Graph Auto-Encoders. arXiv:stat.ML/1611.07308.

[47] Devin Kreuzer, Dominique Beaini, William L. Hamilton, Vincent Létourneau, and Prudencio Tossou. 2021. Rethinking graph transformers with spectral attention. *arXiv preprint arXiv:2106.03893*.
[48] Alex Krizhevsky et al. 2009. Learning multiple layers of features from tiny images. Citeseer.
[49] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[50] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. Retrieved from http://yann.lecun.com/exdb/mnist/.
[51] Ron Levie, Federico Monti, Xavier Bresson, and Michael M. Bronstein. 2017. CayleyNets: Graph convolutional neural networks with complex rational spectral filters. *CoRR* abs/1705.07664.
[52] Bing Li and G. Jogesh Babu. 2019. Convolution theorem and asymptotic efficiency. In *A Graduate Course on Statistical Inference*. Springer New York, New York, NY, 295–327. DOI: https://doi.org/10.1007/978-1-4939-9761-9_10
[53] Hongsheng Li, Guangming Zhu, Liang Zhang, Juan Song, and Peiyi Shen. 2020. Graph-temporal LSTM networks for skeleton-based action recognition. In *Proceedings of the Chinese Conference on Pattern Recognition and Computer Vision (PRCV'20)*. Springer, 480–491.
[54] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*.
[55] Ming Liang, Yao Meng, Jiyu Wang, David L. Lubkeman, and Ning Lu. 2020. FeederGAN: Synthetic feeder generation via deep graph adversarial nets. *IEEE Trans. Smart Grid* 12, 2 (2020), 1163–1173.
[56] Siwu Liu, Ji Hwan Park, and Shinjae Yoo. 2020. Efficient and effective graph convolution networks. In *Proceedings of the SIAM International Conference on Data Mining*. SIAM, 388–396.
[57] Andreas Loukas. 2019. What graph neural networks cannot learn: Depth vs width. *CoRR* abs/1907.03199.
[58] Zhilong Lu, Weifeng Lv, Yabin Cao, Zhipu Xie, Hao Peng, and Bowen Du. 2020. LSTM variants meet graph neural networks for road speed prediction. *Neurocomputing* 400 (2020), 34–45.
[59] Denis Lukovnikov, Jens Lehmann, and Asja Fischer. 2020. Improving the long-range performance of gated graph neural networks. *arXiv preprint arXiv:2007.09668*.
[60] Alessio Micheli. 2009. Neural network for graphs: A contextual constructive approach. *IEEE Trans. Neural Netw.* 20, 3 (2009), 498–511.
[61] Alessio Micheli, Alessandro Sperduti, Antonina Starita, and Anna Maria Bianucci. 2001. Analysis of the internal representations developed by neural networks for structures applied to quantitative structure- activity relationship studies of benzodiazepines. *J. Chem. Inf. Comput. Sci.* 41, 1 (2001), 202–218.
[62] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning distributed representations of graphs. *CoRR* abs/1707.05005.
[63] Chaopeng Pan, Haotian Cao, Weiwei Zhang, Xiaolin Song, and Mingjun Li. 2021. Driver activity recognition using spatial-temporal graph convolutional LSTM networks with attention mechanism. *IET Intell. Transp. Syst.* 15, 2 (2021), 297–307.
[64] Shirui Pan, Ruiqi Hu, Guodong Long, Jing Jiang, Lina Yao, and Chengqi Zhang. 2018. Adversarially Regularized Graph Autoencoder for Graph Embedding. arXiv:cs.LG/1802.04407.
[65] George M. Phillips. 2003. *Interpolation and Approximation by Polynomials*, Vol. 14. Springer Science & Business Media.
[66] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. 2020. GCC: Graph contrastive coding for graph neural network pre-training. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1150–1160.
[67] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. 2020. Karate club: An API oriented open-source Python framework for unsupervised learning on graphs. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM'20)*. ACM.
[68] Luana Ruiz, Fernando Gama, and Alejandro Ribeiro. 2019. Gated graph convolutional recurrent neural networks. In *Proceedings of the 27th European Signal Processing Conference (EUSIPCO'19)*. IEEE, 1–5.
[69] Luana Ruiz, Fernando Gama, and Alejandro Ribeiro. 2020. Gated graph recurrent neural networks. *IEEE Trans. Sig. Process.* 68 (2020), 6303–6318.
[70] Guillaume Salha, Romain Hennequin, and Michalis Vazirgiannis. 2019. Keep It Simple: Graph Autoencoders without Graph Convolutional Networks. arXiv:cs.LG/1910.00942.
[71] Guillaume Salha, Stratis Limnios, Romain Hennequin, Viet-Anh Tran, and Michalis Vazirgiannis. 2019. Gravity-inspired graph autoencoders for directed link prediction. *CoRR* abs/1905.09570.
[72] Peter Sanders and Christian Schulz. 2016. Scalable generation of scale-free graphs. *Inform. Process. Lett.* 116, 7 (2016), 489–491.
[73] Paul-Edouard Sarlin, Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. 2020. SuperGlue: Learning feature matching with graph neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4938–4947.

[74] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. 2009. The graph neural network model. *IEEE Trans. Neural Netw.* 20, 1 (Jan. 2009), 61–80. DOI: https://doi.org/10.1109/TNN.2008.2005605

[75] Franco Scarselli, Sweah Liang Yong, Marco Gori, Markus Hagenbuchner, Ah Chung Tsoi, and Marco Maggini. 2005. Graph neural networks for ranking web pages. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'05)*. IEEE, 666–672.

[76] Marco Serafini. 2021. Scalable graph neural network training: The case for sampling. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 68–76.

[77] David I. Shuman, Sunil K. Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. 2012. Signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular data domains. *CoRR* abs/1211.0053.

[78] David I. Shuman, Sunil K. Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. 2013. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Sig. Process. Mag.* 30, 3 (2013), 83–98.

[79] Chenyang Si, Wentao Chen, Wei Wang, Liang Wang, and Tieniu Tan. 2019. An attention enhanced graph convolutional LSTM network for skeleton-based action recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1227–1236.

[80] Tomas Simon, Hanbyul Joo, Iain A. Matthews, and Yaser Sheikh. 2017. Hand keypoint detection in single images using multiview bootstrapping. *CoRR* abs/1704.07809.

[81] Gerard L. G. Sleijpen and Henk A. Van der Vorst. 2000. A Jacobi–Davidson iteration method for linear eigenvalue problems. *SIAM Rev.* 42, 2 (2000), 267–293.

[82] Ljubisa Stankovic, Danilo P. Mandic, Milos Dakovic, Ilia Kisil, Ervin Sejdic, and Anthony G. Constantinides. 2019. Understanding the basis of graph signal processing via an intuitive example-driven approach [lecture notes]. *IEEE Sig. Process. Mag.* 36, 6 (2019), 133–145.

[83] Shanshan Tang, Bo Li, and Haijun Yu. 2019. ChebNet: Efficient and Stable Constructions of Deep Neural Networks with Rectified Power Units using Chebyshev Approximations. arXiv:cs.LG/1911.05467.

[84] Ke Tu, Peng Cui, Xiao Wang, Philip S. Yu, and Wenwu Zhu. 2018. Deep recursive network embedding with regular equivalence. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'18)*. Association for Computing Machinery, New York, NY, 2357–2366. DOI: https://doi.org/10.1145/3219819.3220068

[85] Rianne van den Berg, Thomas N. Kipf, and Max Welling. 2017. Graph Convolutional Matrix Completion. arXiv:stat.ML/1706.02263.

[86] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR* abs/1706.03762.

[87] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[88] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903*.

[89] Saurabh Verma and Zhi-Li Zhang. 2017. Hunt for the unique, stable, sparse and fast feature learning on graphs. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 88–98. Retrieved from http://papers.nips.cc/paper/6614-hunt-for-the-unique-stable-sparse-and-fast-feature-learning-on-graphs.pdf.

[90] C. Vinchoff, N. Chung, T. Gordon, L. Lyford, and M. Aibin. 2020. Traffic prediction in optical networks using graph convolutional generative adversarial networks. In *Proceedings of the 22nd International Conference on Transparent Optical Networks (ICTON'20)*. IEEE, 1–4.

[91] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. Association for Computing Machinery, New York, NY, 1225–1234. DOI: https://doi.org/10.1145/2939672.2939753

[92] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *Proceedings of the ICLR Workshop on Representation Learning on Graphs and Manifolds*. Retrieved from https://arxiv.org/abs/1909.01315.

[93] Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. 2019. KGAT: Knowledge graph attention network for recommendation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 950–958.

[94] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S. Yu. 2019. Heterogeneous graph attention network. In *Proceedings of the World Wide Web Conference*. 2022–2032.

[95] Xiang Wang, Dingxian Wang, Canran Xu, Xiangnan He, Yixin Cao, and Tat-Seng Chua. 2019. Explainable reasoning over knowledge graphs for recommendation. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 5329–5336.

[96] Oliver Wieder, Stefan Kohlbacher, Mélaine Kuenemann, Arthur Garon, Pierre Ducrot, Thomas Seidel, and Thierry Langer. 2020. A compact review of molecular property prediction with graph neural networks. *Drug Discov. Today: Technol.* 37 (2020), 1–12. DOI: https://doi.org/10.1016/j.ddtec.2020.11.009

[97] Felix Wu, Tianyi Zhang, Amauri H. Souza Jr., Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. 2019. Simplifying graph convolutional networks. *CoRR* abs/1902.07153.

[98] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A comprehensive survey on graph neural networks. *CoRR* abs/1901.00596.

[99] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *CoRR* abs/1810.00826.

[100] Yongqiang Yin, Xiangwei Zheng, Bin Hu, Yuang Zhang, and Xinchun Cui. 2021. EEG emotion recognition using fusion model of graph convolutional neural networks and LSTM. *Appl. Soft Comput.* 100 (2021), 106954.

[101] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. 2021. Do transformers really perform bad for graph representation? *arXiv preprint arXiv:2106.05234*.

[102] Wenchao Yu, Cheng Zheng, Wei Cheng, Charu C. Aggarwal, Dongjin Song, Bo Zong, Haifeng Chen, and Wei Wang. 2018. Learning deep network representations with adversarially regularized autoencoders. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'18)*. Association for Computing Machinery, New York, NY, 2663–2671. DOI: https://doi.org/10.1145/3219819.3220000

[103] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. 2020. Explainability in graph neural networks: A taxonomic survey. *arXiv preprint arXiv:2012.15445*.

[104] Jure Zbontar, Li Jing, Ishan Misra, Yann LeCun, and Stéphane Deny. 2021. Barlow twins: Self-supervised learning via redundancy reduction. *arXiv preprint arXiv:2103.03230*.

[105] Weili Zeng, Juan Li, Zhibin Quan, and Xiaobo Lu. 2021. A deep graph-embedded LSTM neural network approach for airport delay prediction. *J. Adv. Transport.* (2021).

[106] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. 2018. GAAN: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*.

[107] Yiding Zhang, Xiao Wang, Chuan Shi, Xunqiang Jiang, and Yanfang Fanny Ye. 2021. Hyperbolic graph attention network. *IEEE Trans. Big Data*. (2021), 1–1. DOI: 10.1109/TBDATA.2021.3081431

[108] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2018. Deep learning on graphs: A survey. *CoRR* abs/1812.04202.

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

[110] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2018. Graph neural networks: A review of methods and applications. *CoRR* abs/1812.08434.

[111] Yanqiao Zhu, Yichen Xu, Feng Yu, Qiang Liu, Shu Wu, and Liang Wang. 2020. Deep graph contrastive representation learning. *arXiv preprint arXiv:2006.04131*.

[112] Yanqiao Zhu, Yichen Xu, Feng Yu, Qiang Liu, Shu Wu, and Liang Wang. 2021. Graph contrastive learning with adaptive augmentation. In *Proceedings of the Web Conference*. 2069–2080.

Received August 2020; revised October 2021; accepted November 2021