A Practical Guide to Graph Neural Networks

How do graph neural networks work, and where can they be applied?

ISAAC RONALD WARD, ISOLABS[1], Australia
JACK JOYNER[2], ISOLABS, Australia
CASEY LICKFOLD[2], ISOLABS, Australia
STASH ROWE[2], ISOLABS, Australia
YULAN GUO, Sun Yat-sen University, China
MOHAMMED BENNAMOUN, The University of Western Australia, Australia

Graph neural networks (GNNs) have recently grown in popularity in the field of artificial intelligence 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 the average deep learning enthusiast by collating and presenting details on the motivations, concepts, mathematics, and applications of the most common types of GNNs. Importantly, we present this tutorial concisely, alongside worked code examples, and at an introductory pace, thus providing a practical and accessible guide to understanding and using GNNs.

CCS Concepts: • **Theory of computation** → Machine learning theory; • **Mathematics of computing**; • **Computing methodologies** → *Artificial intelligence*; **Machine learning approaches**; **Machine learning algorithms**;

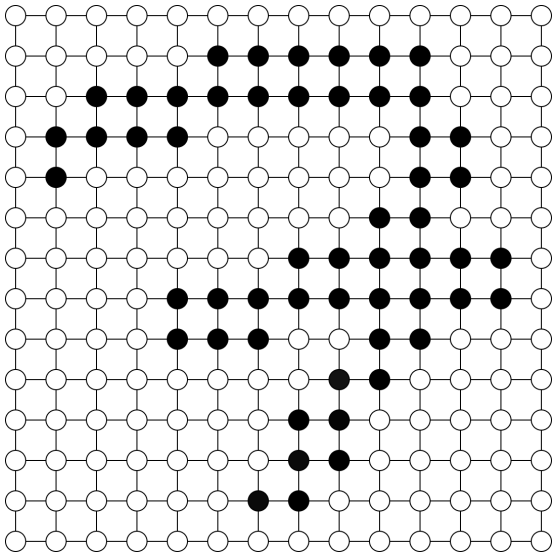
Additional Key Words and Phrases: graph neural network, graph, neural network, tutorial, artificial intelligence, recurrent, convolutional, auto encoder, encoder, decoder, machine learning, deep learning, papers with code, theory, mathematics, applications

1 AN INTRODUCTION

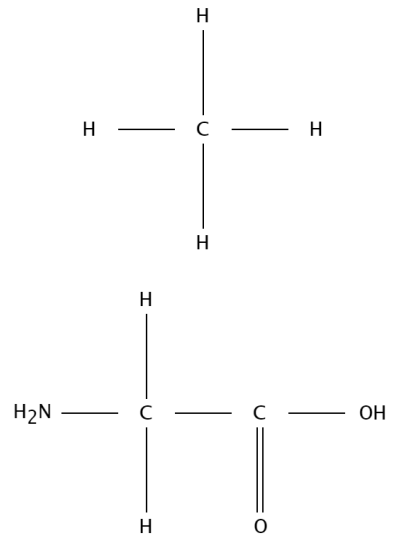
Contemporary artificial intelligence (AI), or more specifically, deep learning (DL) has been dominated in recent years by the learning architecture known as the neural network (NN). NN variants have been designed to increase performance in certain problem domains; the convolutional neural network (CNN) excels in the context of image-based tasks, and the recurrent neural network (RNN) in the space of natural language processing and time series analysis. NNs have also been leveraged as components in composite DL frameworks — they have been used as trainable generators and discriminators in generative adversarial networks (GANs), and as encoders and decoders in transformers [46]. Although they seem unrelated, the images used as inputs in computer vision, and the sentences used as inputs in natural language processing can both be represented by a single, general data structure: **the graph** (see Figure 1).

Formally, a graph is a set of distinct vertices (representing items or entities) that are joined optionally to each other by edges (representing relationships). The learning architecture that has been designed to process said graphs is the titular graph neural network (GNN). 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 [4], a property which makes them valuable in certain 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

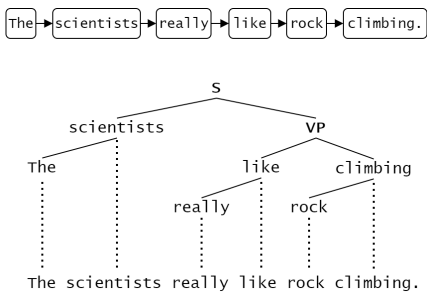
[1] www.isolabs.com.au.
[2] These authors contributed equally to this work.



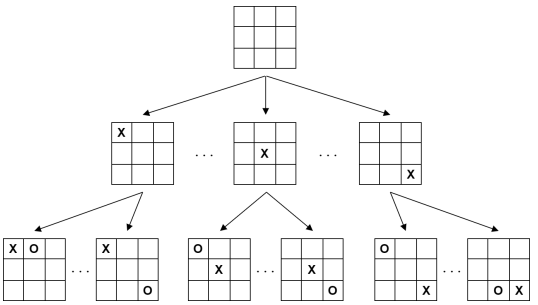
(a) A graph representation of a 14×14 pixel image of the digit '7'. Pixels are represented by vertices and their direct adjacency is represented by edge relationships.



(b) Methane (top) and glycine (bottom) molecular structures represented as graphs. Edges represent electrochemical bonds and vertices represent atomic nuclei or simple molecules.



(c) 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.



(d) 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, etc.

images fed to it must be 28×28 pixels in size to conform to this strict dimensionality requirement [27].

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 comparatively general deep learning methods, and they facilitate the

application of deep learning approaches to sets of data which — until recently — were not possible to work with using traditional NNs or other such algorithms.

1.1 Organisation of this Work

In this tutorial, we have begun with a brief introduction in Section 1. In Section 2 we discuss the key concepts required to understand the operation of GNNs, before looking at specific GNN architectures in Section 3 (recurrent graph geural networks (RGNNs)), Section 4 (convolutional graph neural networks (CGNNs)), and Section 5 (graph auto encoders (GAEs)). In each of these sections, we provide a worked example and provide code repositories to aid the reader in their understanding[3]. This tutorial then concludes in Section 6.

The key contributions of this tutorial paper are as follows:

(1) A concise, easy to understand, introductory tutorial to GNNs.
(2) Explanations of the operation of specific GNN architectures (RGNNs, CGNNs, and GAEs), which progressively build a holistic understanding of the GNN framework (see Sections 3, 4, and 5 respectively).
(3) Fully worked examples of how GNNs can be applied to real world problem domains (see Appendices B.1, B.2, and B.3).
(4) Recommendations for specific further reading and advanced literature (provided at the end of Sections 3, 4, and 5).

We envisage that this work will serve as the 'first port of call' for those looking to explore GNNs and find relevant collated resources, rather than as a comprehensive survey of methods and applications (as in [14, 54, 58, 59]).

2 THE GRAPH DOMAIN

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 which will be used consistently in this work (see Table 1).

2.1 Graph Properties

The core input data structure considered throughout this work is the *graph*. As alluded to earlier, graphs are formally defined as a set of vertices, and the set of edges between these vertices: put formally, $G = G(V, E)$. Fundamentally, remember that 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 Figures 2, 3, and 4 for examples of this domain agnostic expressiveness).

2.1.1 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 N vertices as V and the i^{th} single vertex in the set as v_i. In a graph representing a social network, the vertices might represent people (see Figure 2). Note that there is no requirement for all vertices to be homogenous in their construction.

2.1.2 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 M edges as E and a single edge between the i^{th} and j^{th} vertices as $e_{i,j}$. In a graph

[3]Code will be made public pending current peer review at https://github.com/isolabs/gnn-tutorial. Note that code will be made public pending review.

Notation	Meaning
V	A set of vertices.
N	The number of vertices in a set of vertices V.
v_i	The i^{th} vertex in a set of vertices V.
v_i^F	The feature vector of vertex v_i.
$\mathbf{ne}[v_i]$	The set of vertex *indicies* for the vertices that are direct neighbors of v_i.
E	A set of edges.
M	The number of edges in a set of edges E.
$e_{i,j}$	The edge between the i^{th} vertex and the j^{th} vertex, in a set of edges E.
$e_{i,j}^F$	The feature vector of edge $e_{i,j}$.
h_i^k	The k^{th} hidden layer's representation of the i^{th} vertex's local neighborhood.
o_i	The i^{th} output of a GNN (indexing is framework dependant).
$G = G(V, E)$	A graph defined by the set of vertices V and the set of edges E.
$\mathbf{A}$	The adjacency matrix; each element $\mathbf{A}_{i,j}$ represents if the i^{th} vertex is connected to the j^{th} vertex by a weight.
$\mathbf{W}$	The weight matrix; each element $\mathbf{W}_{i,j}$ represents the 'weight' of the edge between the i^{th} vertex and the j^{th} 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.
$\mathbf{D}$	The degree matrix; a diagonal matrix of vertex degrees or valencies (the number of edges incident to a vertex). Formally defined as $\mathbf{D}_{i,i} = \sum_j \mathbf{A}_{i,j}$.
$\mathbf{L}$	The non-normalized graph Laplacian; defined as $\mathbf{L} = \mathbf{D} - \mathbf{W}$. For unweighted graphs, $\mathbf{W} = \mathbf{A}$.
$\mathbf{L}^{sn}$	The symmetric normalized graph Laplacian; defined as $\mathbf{L} = \mathbf{I}_n - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$.
$\mathbf{L}^{rw}$	The random-walk normalized graph Laplacian; defined as $\mathbf{L} = \mathbf{I}_n - \mathbf{D}^{-1}\mathbf{A}$.
$\mathbf{I}_n$	An $n \times n$ identity matrix; all zero except for one's along the diagonal.

Table 1. Notation used in this work. We suggest that the reader familiarise themselves with this notation before proceeding.

representing a molecular structure, the edges might represent the electrochemical bond between two atoms (see Figure 3).

2.1.3 Features. In AI, features are simply quantifiable attributes which 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) which quantifies the person's age, popularity, and social media usage. Similarly, we might have a feature for each relationship (edge) which 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 $e_{i,j}^F$ respectively.

2.1.4 Neighborhoods. Defined as *subgraphs* within a graph, neighborhoods represent distinct groups of vertices and edges. Neighborhoods can be grown from a single vertex by iteratively

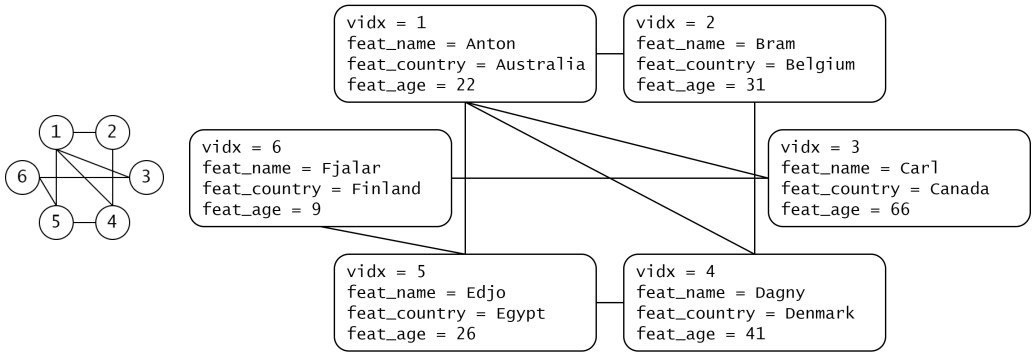


Fig. 2. Two renderings of the same social network graph, one displaying only the edge relationships and vertex indices v_i (left), and one displaying each vertices' vertex features v_i^F (right).

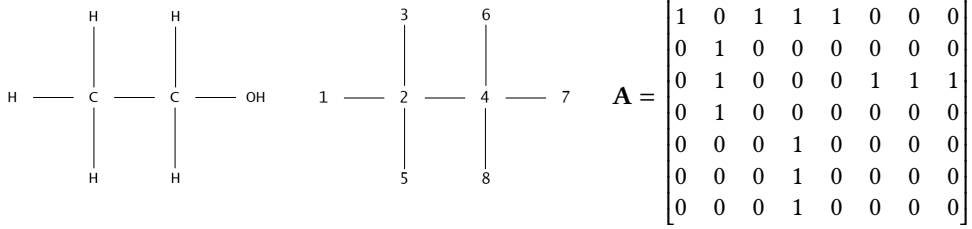


Fig. 3. A diagram of an alcohol molecule (left), its associated graph representation with vertex indices labelled (middle), and its adjacency matrix (right).

considering the vertices attached (via edges) to the current neighborhood. The neighborhood grown from v_i after one iteration is referenced to in this text as the *direct neighborhood*, or by the set of neighbor indices $\mathbf{ne}[v_i]$. Note that a neighborhood can be defined subject to certain vertex and edge feature criteria.

2.1.5 States. Encode the information represented in a given neighborhood around a vertex (including the neighborhood's vertex and edge features, and states). States can be thought of as hidden feature vectors. In practice, states are created by iteratively applying a feature extraction function to the previous iteration's states, with later iteration states including all the information required to perform classification, regression, or some other output calculation on a given vertex.

2.1.6 Embeddings. Defined simply as compressed representations. If we reduce large feature vectors associated with vertices and edges into low dimensional embeddings, 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, neighborhoods, or graphs. Embeddings are also referred to as representations, encodings, latent vectors, or high-level feature vectors depending on the context.

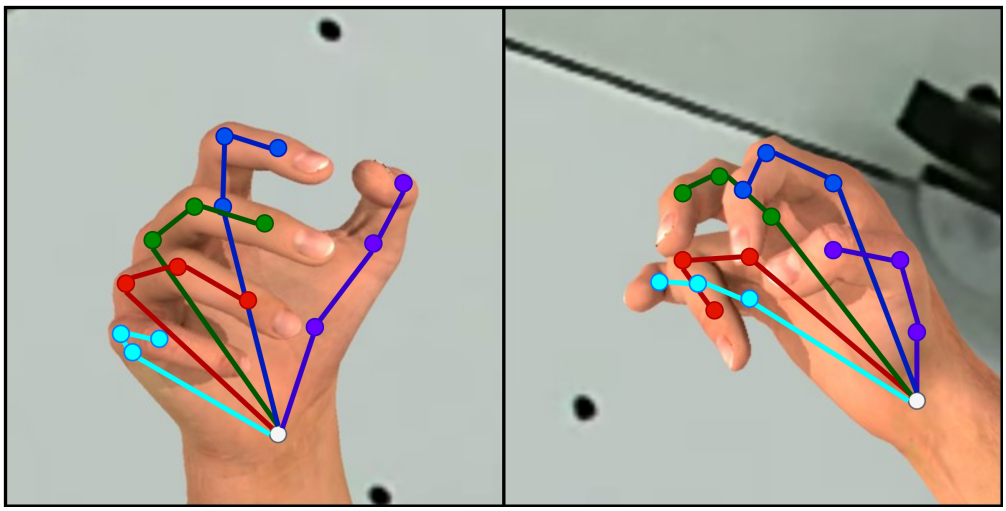


Fig. 4. Two selected frames from a computer generated video of a model hand moving. The rendered graphs show the connections between joints in the human hand, and the hierarchical dependency of said joints. This is actually one **single spatiotemporal** graph — the spatial positions of the vertices in the graph change with respect to time. In practice, these kinds of graphs could be classified into types of hand movements. Images provided from the 'Hands from Synthetic Data' dataset [41]. Image best viewed in colour.

3 RECURRENT GRAPH NEURAL NETWORKS

In a standard NN, successive layers of learned weights work to extract features from an input. In the case of simple image classification, the presence of lower level features — such as short lines and curves — are identified by earlier layers, whereas the presence of higher level and more complex features are identified by later layers. After being processed by sequential layers, the resultant high-level features can then be provided to a softmax layer or single neuron for the purpose of classification, regression, etc.

In this same way, the earliest GNN works aimed to extract high level feature representations from graphs by using successive feature extraction operations [37], and then fed these high level features to output functions. The *recursive* application of a feature extractor, or encoding network, is what provides the RGNN with its name.

3.1 The Forward Pass

The RGNN forward pass occurs in two main steps. **The first step** focuses on computing high level hidden feature vectors for each vertex in the input graph. This computation is performed by a **transition** function, f . **The second step** is concerned with processing the hidden feature vectors into useful outputs; using an **output** function g .

3.1.1 Transition. The transition process considers the neighborhood of each vertex v_i in a graph, and produces a hidden representation for each of these neighborhoods. Since different vertices in the graph might have different numbers of neighbors, the transition process employs a summation over the neighbors, thus producing a consistently sized vector for each neighborhood. This hidden representation is often referred to as the vertex's state [37], and it is calculated based on the following quantities.

(1) v_i^F — the features of the vertex v_i, which the neighborhood is centered around.

(2) $e_{i,j}^F$ — the features of the edges which join v_i to its neighbor vertices v_j. Here only direct neighbors will be considered, though in practice neighbors further than one edge away may be used. Similarly, for directed graphs, neighbors may or may not be considered based on edge direction (e.g., only outgoing or incoming edges considered as valid neighbor connection).
(3) v_j^F — the features of v_i's neighbors.
(4) h_j^{k-1} — the previous state of v_i's neighbors. Recall that a state simply encodes the information represented in a given neighborhood.

Formally, the transition function f is used in the recursive calculation of a vertex's k^{th} state as per Equation 1.

$$h_i^k = \sum_{j \in \mathbf{ne}[v_i]} f(v_i^F, e_{i,j}^F, v_j^F, h_j^{k-1}), \text{where all } h_i^0 \text{ are defined upon initialisation.} \tag{1}$$

We can see that under this formulation, f is well defined. It accepts four feature vectors which 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 recursively, until a stable state is reached for all vertices in the input graph. If f is a contraction map, Banach's fixed point theorem ensures that the values of h_i^k will converge to stable values exponentially fast, regardless of the initialisation of h_i^0 [21].

The iterative passing of 'messages' or states between neighbors to generate an encoding of the graph is what gives this *message passing* operation its name. In the first iteration, any vertex's state encodes the features of the neighborhood within a single edge. In the second iteration, any vertex's state is an encoding of the features of the neighborhood within two edges away, and so on. This is because the calculation of the k^{th} state relies on the $(k-1)^{th}$ state. To fully elucidate this process, we will step through how the transition function is recursively applied (see Figures 5, 6, and 7).

The purpose of repeated applications of the transition function is thus to create discriminative embeddings which can ultimately be used for downstream machine learning tasks. This is investigated further via a worked example in Appendices B.1, and we recommend that readers look through the example (and code) if further explanation is required.

3.1.2 Output. The **output** function is responsible for taking the converged hidden state of a graph $G(V, E)$ and creating a meaningful output. Recall that from a machine learning perspective, the repeated application of the transition function f to the features of $G(V, E)$ ensured that every final state h_i^{kmax} is simply an encoding of some region of the graph. The size of this region is dependent on the halting condition (convergence, max time steps, etc.), but often the repeated 'message passing' ensures that each vertex's final hidden state has 'seen' the entire graph. These rich encodings typically have lower dimensionality than the graph's input features, and can be fed to fully connected layers for the purpose of classification, regression, and so on.

The question now is what do we define as a 'meaningful output'? This depends largely on the task framework:

- **Vertex-level frameworks**. A unique output value is required for every vertex, and the output function g takes a vertex's features and final state as inputs: $o_i = g(v_i^F, h_i^{kmax})$. The output function g can be applied to every vertex in the graph.
- **Edge-level frameworks**. A unique output value is required for each edge, and the output function g takes the edge's features, and the vertex features and final state of the vertices which define the edge: $o_{ij} = g(e_{i,j}^F, v_i^F, h_i^{kmax}, v_j^F, h_j^{kmax})$. Similarly, g can be applied to every edge in the graph.

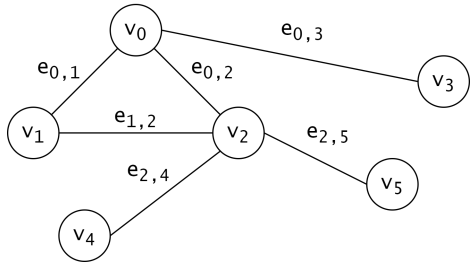


Fig. 5. A simple graph with $N = 6$ and $M = 6$ is defined using the notation outlined in Table 1. It is understood that every vertex and edge has some associated features. Additionally, each vertex v_i is initialised with some h_i^0.

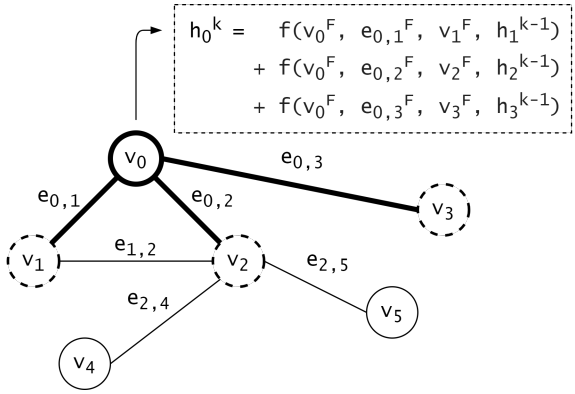


Fig. 6. We calculate the k^{th} state for each neighborhood in the graph. Note that any k^{th} state calculation is dependent on only the input graph features of the graph and on the $(k-1)^{th}$ states. We begin by calculating h_0^k for the neighborhood centered around v_0, as per Equation 1. v_0 and its direct connections (emboldened) to its neighbor vertices (dotted) are illustrated here.

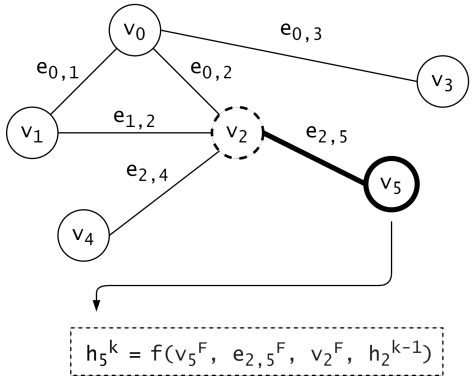


Fig. 7. The same calculation is performed on each neighborhood in the graph, until the final vertex's neighborhood is reached. Once this is done, the k^{th} layer representation of the graph has been calculated. The process is then repeated for the $(k+1)^{th}$ layer, the $(k+2)^{th}$ layer, and so on, typically until convergence is reached.

- **Graph-level frameworks**. A unique output is required over the entire graph (i.e., graph classification). The output function $g$will consider all relevant information calculated thus far. This will include the final state around each vertex, and may optionally include the initial vertex and edge features: $o = g(h_0^{kmax}, h_1^{kmax}, ..., h_{N-1}^{kmax}, v_0^F, v_1^F, ..., v_{N-1}^F, e_{0,0}^F, e_{0,1}^F, ...e_{0,N-1}^F, ..., e_{N-1,N-1}^F)$. Here, g is only applied once per graph.

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 [59]. This is possible because of a loss function which defines an error between the predicted output and a labelled ground truth. Both f and g can then be trained via backpropagation of errors. For more detail on this process, see the calculations in [37].

3.2 In Summary of Recurrent Graph Neural Networks

In this section, we have explained the forward pass that allows a 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 vertex. The repeated application of f ensures that at iteration k, a given vertex's state includes information from neighbor vertices k edges away. These high-level features can be fed to an output function to solve a given task. During the backward pass, the parameters for the NNs f and g are updated with respect to a loss which is backpropagated through the layers of computation performed during the forward pass. Ultimately, understanding RGNNs is a stepping stone in understanding more complex GNNs and GNN-based models.

3.2.1 Further Reading. In one of the earlier investigations of GNNs [37], Scarselli et al. investigated how an algorithm, which parses a graph input into an output of a desired shape, might be trained. Scarselli et al. outline the design and constraints on the forward pass operations, and specify the mathematical foundations for backward pass algorithms, ultimately providing an in-depth technical discussion on the formulation, operation, and computation complexities associated with an RGNN.

In [4], Bronstein et al. introduce the term *geometric deep learning* to describe deep learning methods which are applicable to non-Euclidean domains such as graphs, point clouds, meshes, and manifolds. They classify a number of algorithms (including GNNs) as geometric deep learning algorithms, and go on to contrast and compare the differences in operation between Euclidean and non-Euclidean deep learning algorithms. Bronstein et al. discuss the applications of geometric deep learning algorithms to a variety of specific problem domains (e.g., computer graphics).

Hu et al. [18] propose a novel strategy for pre-training GNNs, and thus explore how vertex embeddings can be made to be more generalizable. Importantly, they pre-train GNNs at both the local (vertex) and global (graph) levels, thus ensuring that local and global representations can be learned in tandem. Conversely, pre-training strategies which focus on either only local or only global are found to result in limited performance gains or worse performance in downstream tasks. Specific GNN architectures which benefit the most from pre-training are also highlighted .

Loukas focuses on investigating what GNNs can and cannot learn [30], providing experimental results concerning the parameter choices for the GNN architecture and the effect that these have on a GNN's performance. Loukas' investigation is restricted to GNNs which use message passing during their propagation step (or forward pass).

Haan et al. address a key shortcoming found in RGNNs and GNNs in general: the message aggregation process found in the forward pass is invariant to the order of the messages / embeddings (see Equation 1), and because of this, GNNs 'forget' how information flows throughout a graph [8]. Haan et al. propose an *equivariant* solution to this problem in natural graph networks (NGNs).

NGNs weights are dependent on the local structure of the graph — as opposed to methods that attempt to use the global structure of the graph and thus incur large computational costs. As such, the messages which are passed in an NGN's forward pass include features that are sensitive to the flow of information over the graph. Haan et al. find that such NGN frameworks provide competitive results in a variety of tasks, including graph classification.

4 CONVOLUTIONAL GRAPH NEURAL NETWORKS

CGNNs are to RGNNs as CNNs are to NNs. In fact, CGNNs were largely inspired by the success of CNNs on image-based tasks in computer vision — by applying convolutional operations, CGNNs are able to efficiently perform hierarchical pattern recognition within the graph domain. Although the transition process during a CGNN's forward pass is altered to include the eponymous convolution operation, the core high-level process remain the same as in RGNNs.

4.1 Spatial Convolution

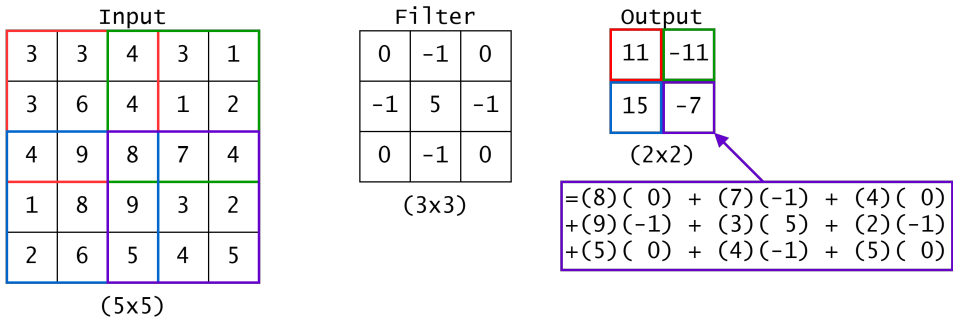


Fig. 8. A convolutional operation of 2D matrices. This process is used in computer vision and in CNNs. The convolutional operation here has a stride of 2 pixels. The given filter is applied in the red, green, blue, and then purple positions — in that order. At each position each element of the filter is multiplied with the corresponding element in the input 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 CNNs, the filter is learned during training, so that the most discriminative features are extracted while the dimension is reduced. In the case of this image the filter is a standard sharpening filter used in image analysis. Image best viewed in colour.

When we think of convolution, we often think of the convolution operation used in CNNs (see Figure 8). Indeed, this conforms to a general definition of convolution:

> "An output derived from **two** given inputs by integration (or summation), which expresses how the shape of 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. In the spatial graph domain, it seems that this type of convolution is not well defined [40]; 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 formal definition of convolution is 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 or kernels can be utilised 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. Relating to our previous example in CNNs, the learned convolutional filters perform a kind of learned feature extraction.

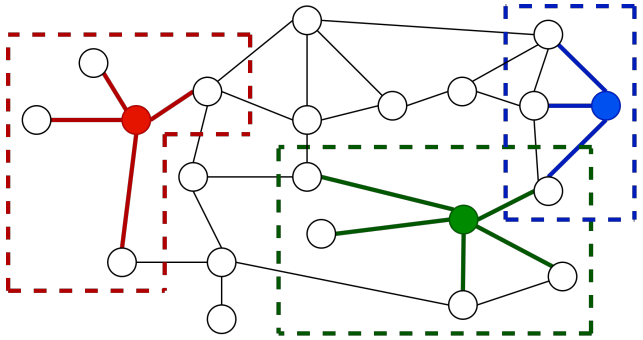


Fig. 9. Three neighborhoods in a given graph (designated by dotted boxes), with each one defined by a central vertex (designated by a correspondingly coloured circle). In spatial convolution, a neighborhood is **selected**, the values of the included vertices' features are **aggregated**, and then this aggregated value is used to **update** the vertex embedding of the central vertex. This process is **repeated** for all neighborhoods in the graph. These embeddings can then be used as vertex 'values' in the next layer of spatial convolution, thus allowing for hierarchical feature extraction. In this case a vertex is a neighbor of another vertex only if it is directly connected to it. Note the similarities to the transition process of RGNNs, as illustrated in Figures 5, 6, and 7. Image best viewed in colour.

Consider the similarities between images and graphs (see Figure 1). Images are just *particular cases* of graphs: pixels are vertices, and edges exist between adjacent pixels [54]. Image convolution is then just a specific example of graph convolution. One full graph convolution operation is then completed as follows (see Figure 9 for an illustrated example). Note that we use the term spatial connectivity to describe if two vertices are connected by an edge (or 'connected' under some other spatial definition).

(1) Using *spatial connectivity* to define graph neighborhoods around all vertices, **select** the first neighborhood in the input graph.
(2) **Aggregate** the values in this neighborhood (e.g., using a sum or mean calculation).
(3) Use the aggregated values to **update** the central vertex's hidden state.
(4) **Repeat** this process on the subsequent neighborhoods in the input graph.

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 [13], 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. GraphSAGE has since been outperformed on accepted benchmarks [10] by other frameworks [3], but the framework is still competitive and can be used to explore the concept of learned aggregators (see Appendices B.2).

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. The hidden states

calculated during the first pass of spatial graph convolution can be considered in the next stage of spatial graph convolution, thus allowing for increasingly higher level feature extraction. This operation propagates vertex information along edges in a similar manner to RGNNs (Section 3).

Interestingly, at this point we can see that there are many similarities to RGNNs and spatial CGNNs. The core difference is that RGNNs continue iteration until a stable point, and use a repeated transition function in doing so. Alternatively, CGNNs iterate for a fixed number of layers, each of which usually have only a single layer of weights being applied.

4.2 Graph Signal Processing

Before we discuss the mechanics of spectral convolutions in the graph domain, we must first familiarise ourselves with elements of graph signal processing. A *signal* is defined on a graph if it maps every vertex in a given graph to a real value. Formally, a function f is considered a signal if $f : V \rightarrow \mathbb{R}, \forall\ V \in G$. To simplify this, it may be beneficial to think of a signal simply as a type of feature which every vertex possesses — this could easily be presented as a vector whose i^{th} element represents the signal value at v_i (see Figure 10).

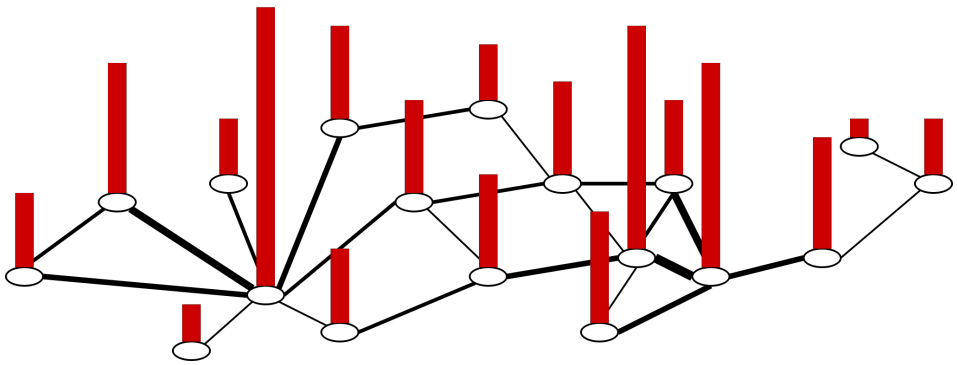


Fig. 10. Pictured is an example graph $G(V, E)$ rendered in the plane ($N = 18$, $M = 23$), with corresponding graph signal values rendered as red bars rising perpendicularly to said plane. The vertices V represent cities, and the edges E represent if two cities have a flight path between them. This graph also has a weight matrix $\mathbf{W}$ signifying the average number of flights per week along a given flight path which are actually conducted. The weight / number of flights associated with each edge / flight path is proportional to the edge's thickness. The graph signal represents the number of people with a contagious virus in each city, since there are 18 vertices, the graph signal vector is accordingly of size 18. Other graphs and graph signals can be defined for neural networks, traffic networks, energy networks, social networks, etc [39, 42].

For example, a graph might have vertices that represent cities, and weighted edges that represent the amount of average weekly flights which go between two cities. In this context, we could have a signal which represents the amount of people with a contagious virus in each city (see Figure 10). As previously mentioned, we could describe this scenario with a different language; each vertex has a feature which is the number of people with a contagious virus in the city which that vertex represents. Again, this *collective* set of features (or samples) is referred to as the graph signal [39].

4.2.1 The Mathematics. Unfortunately many of the simple and fundamental tools for understanding and analysing a signal are not well defined for graphs that are represented in *vertex space*. For example, the translation operation has no immediate analogue to a graph [39]. Similarly, there is no immediate way to convolve a graph directly with a filter. To overcome the challenges in processing

signals on graphs, the inclusion of spectral analysis — similar to that used in discrete signal analysis — has been fundamental in the emergence of localised computation of graph information. To process graph signals in this way we are required to transform the graph from vertex space to frequency space. It is within this frequency space that our previous understandings of mathematical operations (such as convolution) now hold.

We leverage Fourier techniques to perform this transformation to and from the vertex and frequency spaces. Classically, the Fourier transform is the expansion of a given function in terms of the eigenfunctions of the Laplace operator [39]. In the graph domain, we use the *graph* Fourier transform, which is the expansion of a given *signal* in terms of the eigenfunctions of the *graph* Laplacian (see Table 1 for a formal definition of the graph Laplacian, and see Figure 11 for a concrete example).

The graph Laplacian $\mathbf{L}$ is a real symmetric matrix, and as such, we can perform eigendecomposition on it and hence extract its set of k eigenvalues λ_k and eigenvectors u_k (its eigensystem). In reality, there are many ways to calculate a matrix's eigensystem, such as via singular value decomposition (SVD). Additionally, other graph Laplacian's have been used, such as $\mathbf{L}^{sn}$ and $\mathbf{L}^{rw}$ (see Table 1).

By the fundamental property of eigenvectors, the Laplacian matrix can be factorised as three matrices such that $\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$. Here, $\mathbf{U}$ is a matrix with eigenvectors as its columns, ordered by eigenvalues, and $\mathbf{\Lambda}$ is a diagonal matrix with said eigenvalues across its main diagonal. Interestingly, the eigenvectors of $\mathbf{L}$ are exactly the exponentials of the discrete Fourier transform.

Once this eigendecomposition has been performed and the eigensystem has been calculated, we can freely express a given discrete graph signal f in terms of the eigenvectors of the graph Laplacian, thus producing the graph Fourier transform $\hat{f}$. This is performed using Equation 2, and the inverse transformation is performed using Equation 3. Again, the k eigenvectors u_k appear as columns in the matrix $\mathbf{U}$, which is itself generated when calculating the eigensystem of the graph Laplacian matrix $\mathbf{L}$.

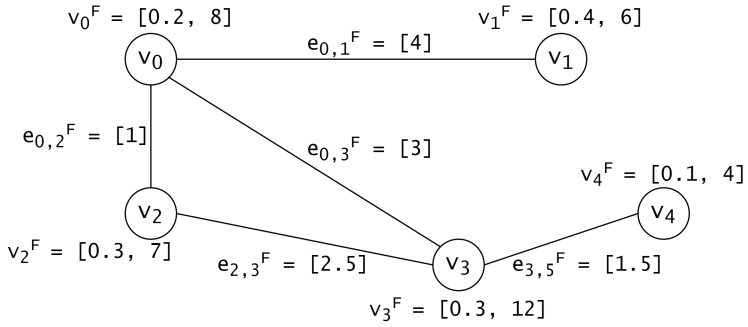
$$\hat{f}(k) = \sum_{i=1}^{N} f(i)\, u_k^*(i) \quad \text{or} \quad \hat{f} = \mathbf{U}^T f \tag{2}$$

$$f(i) = \sum_{k=1}^{N} \hat{f}(k)\, u_k(i) \quad \text{or} \quad f = \mathbf{U}\, \hat{f} \tag{3}$$

As per convolution theorem, multiplication in the frequency domain corresponds to convolution in the time domain [29]. This property is also valid for graphs: multiplication in the vertex domain is equivalent to convolution in the frequency domain. This is an important property, as we cannot define the classical convolutional operator directly in the vertex domain, since classical convolution requires a translation operator — one signal is imposed on the other at every possible position. However, we can define convolution in the frequency domain as per Equation 4.

$$(f * g)(i) = \sum_{k=1}^{N} \hat{f}(k)\, \hat{g}(k)\, u_k(i) \tag{4}$$

In practice, the function g that appears in Equation 4 is typically a learned filter (as alluded to in Section 4.1), but we will cover this more specifically in Section 4.3. For worked examples and an in depth explanation of graph signal processing we recommend the following works: [39, 42]. With an understanding of how convolution can be performed on graphs in the spectral domain, we can now delve into the mechanics of spectral convolution.



$$\mathbf{L} = \mathbf{D} - \mathbf{A} = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 3 & -1 & -1 & -1 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 2 & 1 & 0 \\ -1 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$$

Fig. 11. A concrete example of how the graph Laplacian is formed for a graph, and how we might represent edge weights and graph signals. In this example, there are two graph signals; the first graph signal $f_1 = [0.2, 0.4, 0.3, 0.3, 0.1]$, equivalently, it is the ordered set of the first elements from each vertices' feature vector v_i^F. Similarly, the second graph signal is $f_2 = [8, 6, 7, 12, 4]$. The graph Laplacian in this example is simply formed by $\mathbf{L} = \mathbf{D} - \mathbf{A}$, and it is both real and symmetric, thus it can be decomposed to determine its eigensystem (again, definitions of mathematical notation are available in Table 1).

4.3 Spectral Convolution

Motivated by the abundance of work in signal analysis and graph signal processing, *spectral convolution* was developed as another method for generating hidden layer representations for graphs. The method leverages the transformations defined in Section 4.2 to convert the previous layer's hidden representation into a mathematical space where the eigenvectors of the normalised graph Laplacian define a basis. In this space, a **learned filter** is applied to the **previous layer's hidden output**, before it is transformed back into its original space (note that the application of a filter to an input meets our earlier definition of a convolution operation).

Before we can perform a single layer of spectral convolution, we must determine the graph Laplacian's eigensystem (as described in Section 4.2), as this allows us to perform transforms between vertex and frequency space. This is typically an $O(N^3)$ complexity operation. With the eigensystem in hand, we can now convolve a single graph signal f with a learned filter Θ, by first converting f into the frequency space, and then multiplying it with Θ (as per Equation 4). Mathematically, this is formulated as $\Theta(\mathbf{U}^T f)$. As training progresses, this learned filter begins to extract progressively more discriminative features.

In order to repeat this process recursively, we need to return the convolved result into the vertex space, so the operation becomes $\mathbf{U}(\Theta(\mathbf{U}^T f))$. Note the dimensionality here; both $\mathbf{U}$ and $\mathbf{U}^T$ have dimensions of $(N \times N)$, the signal f by definition must map every vertex to a real value (see Section 4.2), so it has dimension $(N \times 1)$. Thus, the learned filter's dimension must too be $(N \times N)$.

In practice, a graph might have multiple features per vertex, or in other words, multiple graph signals (see Section B.3 for an example). We define f_k as the number of graph signals (or hidden channels) in the k^{th} layer. We also define the k^{th} layer's hidden representation of the i^{th} graph signal's element belonging to the j^{th} vertex as $H^k_{j,i}$. The goal is then to aggregate each of the f_k graph signals like so $\sum_i^{f_k} \mathbf{U}(\Theta(\mathbf{U}^T H^k_{:,i}))$, where the ':' symbol designates that we are considering every vertex.

The spectral convolution process is now almost complete. We firstly need to add non-linearity to each layer of processing in the form of an activation function σ. Secondly, we must define our forward pass with respect to subsequent layers of processing. This forms Equation 5, the forward pass for spectral convolution. Note that we are calculating $H^k_{j,:}$ — the k^{th} layer's j^{th} hidden channel for every vertex — and we do this for each of the f_k hidden channels.

$$H^k_{j,:} = \sigma(\sum_{i=1}^{f_{k-1}} \mathbf{U}(\Theta^k(\mathbf{U}^T H^{k-1}_{:,i}))), \quad \text{where } j = 1, 2, ..., f_k \tag{5}$$

4.3.1 Limitations and Improvements. There are some obvious shortcomings with this basic form of spectral convolution, not the least of which is having to recalculate the eigensystem if the graph's structure is altered. This requirement precludes the use of spectral methods on dynamic graphs, and also means that the learned filters Θ^k are domain dependant — they only work correctly on the graph structure they were trained on [54].

These initial limitations were alleviated by **ChebNet**, an approach which approximates the filter Θ^k using Chebyshev polynomials of $\mathbf{\Lambda}$ [22, 43]. This approach ensures that filters extract local features independently of the graph. Furthermore graph convolutional networks (or **GCNs**) further reduce the computational complexity of determining the eigensystem from $O(N^3)$ to $O(M)$ by introducing first-order approximations and normalization tricks to improve numerical stability [28, 53, 54].

Interestingly, spectral methods which use order K polynomials as their filters (such as ChebNet and GCNs [22, 28, 43]) can be shown to be producing linear combinations of input signals at vertices with a K-hop local neighborhood [39]. This allows us to interpret these spectral filtering methods as spatial convolution methods, thus linking spectral convolution methods back to spatial convolution methods.

4.4 In Summary of Convolutional Graph Neural Networks

In this section we have detailed the two key approaches to convolution in the graph domain: spatial and spectral. In doing so, we have looked at how spatial convolution inherits mechanics from the original RGNNs, and at how spectral convolution inherits mechanics from graph signal processing. Ultimately, these two approaches to convolution have many equivalencies, and together they define the forward passes for the body of algorithms which we refer to as CGNNs.

4.4.1 Further Reading. Wu et al. conduct one of the first formal survey on GNNs in [54]. Their survey classifies both open source resouces and publications on GNNs into four main categories of GNN: RGNNs, CGNNs, GAEs, and Spatial-temporal GNNs. Although each category is discussed in detail, there is a specific focus on CGNNs. The differences in performance, inputs, and design between spatial based and spectral based CGNNs is presented with reference to the last few years of impactful GNN research.

In [10], Dwivedi et al. provide a framework for benchmarking GNNs, and subsequently benchmark all of the most popular current CGNNs on a multitude of different datasets (ZINC [20], MNIST [26], CIFAR10 [24], etc.) and prediction frameworks (vertex classification, edge prediction, graph

classification, and graph regression). A focus is placed on identifying medium-scale datasets which can statistically separate GNN performances, and all experimental parameters are explicitly defined. A variety of general findings are provided based on the empirical results of said experiments. For example, it is found that graph agnostic NNs perform poorly, i.e., multi-layer perceptron algorithms which update vertex embeddings independently of one another (without considering the graph structure) are not suitable for tasks in the graph domain.

Zhang et al. discuss all major methods of deep learning in the graph domain [58], and as such include a discussion on GNNs. Uniquely, Zhang et al. include detailed discussions on Graph Reinforcement Learning and Graph Adverserial Methods, though there is again a focus on CGNNs. As in [54], a distinction is drawn between vertex level and graph level learning frameworks.

In [59], Zhou et al. provide a GNN taxonomy, and distinctly classify GNNs based on the type of graph ingested, the method of training, and the method of propagation (during the forward pass). Numerous aggregation and updater functions are formally described, compared, and classified. Additionally, GNNs are also classified by problem domain; applications to knowledge graphs, physical systems, combinatorial optimisation problems, graph generation, as well as text and image based problems are considered.

In general, there has been much discussion in the literature on improving GCNs: a stand-alone GCN was shown by [7] to have poor performance, particularly on graph-level classification tasks. It was supposed that this was due to the *isotropic* nature of the edges (all edges are weighted equally). More accurate algorithms take an *anisotropic* approach, such as Graph Attention Networks (GAT) [47] and GatedGCN [3]. If the edges are weighted differently, the importance of vertices and the state of the graph can be 'remembered'. The covalent single and double bonds in a molecular application of GNNs would be an example of an anisotropic architecture. Other applications make use of residual connections, such as the Graph Isomorphism Network (GIN) [55]. The residual connections allow GIN to use the learned features from all layers in the prediction. GIN also makes use of batch normalization [19], but in the graph domain.

5 GRAPH AUTOENCODERS

Graph autoencoders (GAEs) represent the application of GNNs (often CGNNs) to autoencoding. Perhaps more than other machine learning architecture, autoencoders (AEs) transition smoothly to the graph domain because they come 'prepackaged' with the concept of *embeddings*. In their short history, GAEs have lead the way in unsupervised learning on graph-structured data and enabled greater performance on supervised tasks such as vertex classification on citation networks [23].

5.1 Traditional Autoencoders

AEs are a common unsupervised learning technique, with successful applications in a variety of tasks including image denoising [49], dimensionality reduction [17] and recommender engines [38]. They work in a two-step fashion, first encoding the input data into a latent space, thus reducing its dimensionality, and then decoding this compressed representation to reconstruct the original input data, as depicted in Figure 12 (though in some cases higher dimensionality latent space representations have been used).

As stated, the AE is trained to minimise 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 measured as the mean squared error between the input instance X and the reconstructed input $\hat{X}$, as per Equation 6.

$$\text{Loss}_{AE} = \|X - \hat{X}\|^2 \tag{6}$$

We can perform end-to-end learning across this network to optimize the encoding and decoding in order to strike a balance between both **sensitivity** to inputs and **generalisability** — we do not want the network to overfit and 'memorise' all training inputs. Additionally, the latent space representations of an encoder-decoder network can be referred to as the *embedding* of the inputs, and they are analogous to the vertex embeddings which were discussed in earlier sections. In other words, a latent representation is simply a learned feature vector.

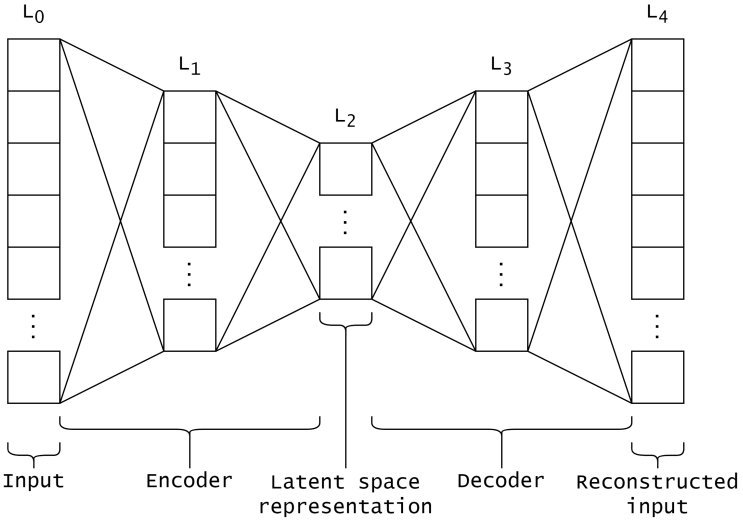


Fig. 12. The architecture for a very simple standard AE. AEs take an original input, alter the dimensionality through multiple fully connected layers, and thus convert said input into a latent space vector (this process forms the encoder). From there, the AE attempts to reconstruct the original input (this process forms the decoder). By minimising the reconstruction loss (see Equation 6), efficient latent space representations can be learned. This diagram shows a AE with a latent space representation that is smaller than the input size, such that $\dim(L_0) = \dim(L_4) > \dim(L_1) = \dim(L_3) > \dim(L_2)$. In practice, other layers such as convolutional layers can be used in lieu of fully connected layers.

5.1.1 Variational Autoencoders. Rather than representing inputs with single points in latent space, variational autoencoders (VAEs) learn to encode inputs as probability distributions in latent space. 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 generalisation to unseen inputs and regularising the model to avoid overfitting. Ultimately, this approach transforms the AE into a more suitable generative model.

Figure 13 shows a VAE which predicts a normal distribution $N(\mu, \sigma)$ for a given input (in practice, the mean and covariance matrix are typically used to define these normal distributions). Unlike in AEs — where the loss is simple the mean squared error between the input and the reconstructed input — a VAEs loss has two terms. This is shown formally in Equation 7. The first term is the reconstruction term we saw in Equation 6, this term still appears as its obviously still our desire to accurately reconstruct the provided input. The second term regularises the latent space distributions by ensuring that they do not diverge significantly from standard normal distributions (denoted as $N(0, 1)$), using **Kulback-Leibler divergence** (denoted as KL in Equation 7). Without this second term, the VAE might return distributions with small variances, or high magnitude means: both of which would cause overfitting.

$$\text{Loss}_{VAE} = \|X - \hat{X}\|^2 + \text{KL}(N(\mu, \sigma), N(0, 1)) \tag{7}$$

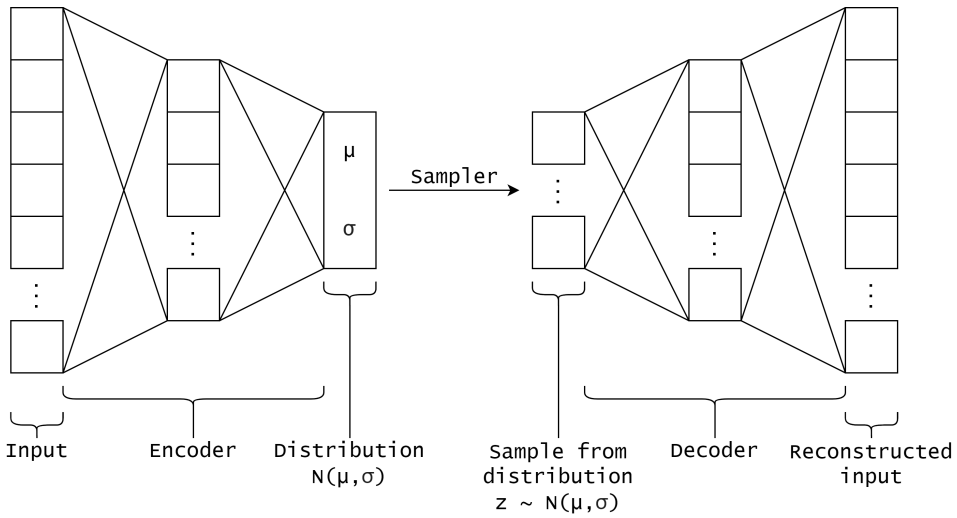


Fig. 13. This diagram depicts a simple VAE architecture. The encoder portion of this VAE takes an input, and encodes it via multiple fully connected layers into a distribution in latent space. This distribution is then sampled to produce a latent space vector (akin to the latent space representation in Figure 12). The decoder portion of this VAE takes this latent space vector, and decodes it into a reconstructed input as in an AE. Equation 7 illustrates the loss term which is minimised in a VAE architecture.

5.2 Application to Graphs

AEs translate well to the graph domain because they are designed around the concept of learning input embeddings. As with AEs, GAEs can be used to train encoders to learn discriminative features in an unsupervised manner, and these encoders can then be used to perform specific downstream machine learning tasks, such as vertex classification or clustering [2, 23, 33, 56]. Indeed, GAE architectures commonly measure their success by performance on such supervised tasks. Moreover, GAEs are useful tools for exploring and understanding graph-structured data by creating meaningful representations of the graph in lower, visualisable dimensions (2D and 3D).

5.2.1 The Target Variable. In AEs, the target input has a well defined structure (e.g., a vector of known length, or an image of known size), and thus the quality of its reconstruction is easily measured using mean squared error. To measure the suitability of a reconstruction in a GAE, we need to produce a meaningfully structured output that allows us to identify similarities between the reconstruction and the input graph.

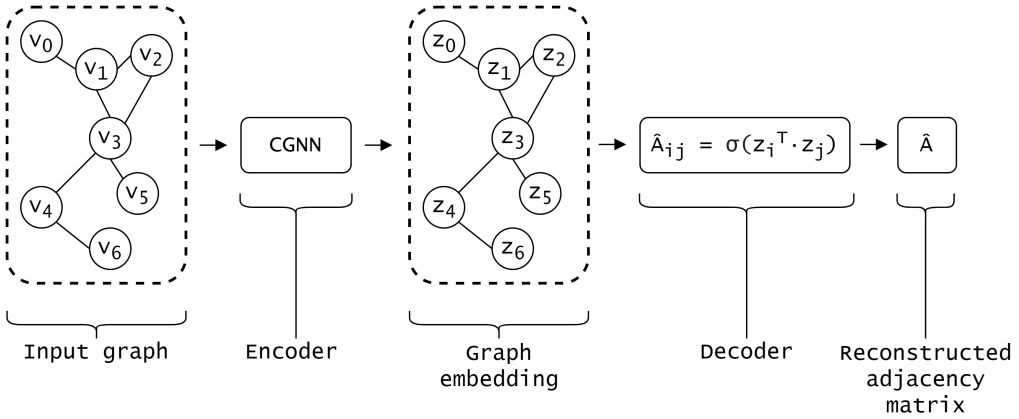


Fig. 14. This diagram demonstrates the high level operation of a basic GAE. Rather than ingesting an matrix or vector, the encoder ingests graph structured data G. In this case a CGNN fulfills the role of the encoder by creating discriminative vertex features / vertex embeddings for every vertex. This does not alter the structure of G. As with AEs and VAEs, the decoder's role is to take these embeddings and create a datastructure which can be compared to the input. In the case of GAEs, G's adjacency matrix is reconstructed as per Section 5.2.4. This reconstructed adjacency matrix $\hat{A}$ (see Equation 8) is compared with the original adjacency matrix A to create a loss term, which is then used to backpropagate errors throughout the GAE and allow training.

5.2.2 The Loss Function. In VGAEs the loss is similar to the standard VAE loss; KL-divergence is still used to measure similarity between the predicted and true distributions, however now binary cross-entropy is used to measure the difference between the predicted adjacency matrix and true adjacency matrix, which replaces the reconstruction component of standard AE loss. Alternative methods include using distance in Wasserstein space (this is known as the Wassertein metric, and it quantifies the cost associated with making a probability distribution equal to another) rather than KL-divergence [60], L2-reconstruction loss, Laplacian eigenmaps, and the ranking loss [2].

5.2.3 Encoders. Before the seminal work in [23] on VGAEs, a number of deep GAEs had been developed for unsupervised training on graph-structured data, including Deep Neural Graph Representations (DNGR) [6] and Structure Deep Network Embeddings (SDNE) [50]. Because these methods operate on only the adjacency matrix, information about both the entire graph and the local neighbourhoods was lost. More recent work mitigates this by using an encoder that aggregates infromation from a vertex's local neighbourhood to learn latent vector representations. For example, [35] proposes a linear encoder that uses a single weights matrix to aggregate information from each vertex's one-step local neighbourhood, showing competitive performance on numerous benchmarks. Despite this, typical GAEs use more complex encoders — primarily CGNNs — in order to capture nonlinear relationships in the input data and larger local neighbourhoods [2, 6, 23, 33, 44, 45, 50, 56].

5.2.4 Decoders. Standard GNNs, regardless of their aggregation functions or other details, typically generate a matrix containing the embedding of each vertex. GAEs apply a decoder to this encoded representation of each vertex in order to reconstruct the original graph. For this reason, the decoder aspect of GAEs is commonly referred to as the *generative model*. In many variations of GAEs [23], the decoder is a simple inner product of the latent variables. As an inner product of two vectors is equivalent to calculating their cosine similarity, the higher the result of this product the more likely are those vertices connected. With this prediction of vertex similarity, the graph's adjacency matrix can be predicted solely from the embeddings of all of the vertices in the graph.

More formally, the encoder produces a vertex embedding z_i for each vertex v_i, and we then compute the cosine similarity for each pair (z_i, z_j) which gives us an approximation for the distance between v_i and v_j (after adding non-linearity via an activation function σ). By computing all pairwise distance approximations, we have produced an approximation of the adjacency matrix $\hat{A}$. This can be compared to $\mathbf{A}$ to determine a reconstruction error, which can then be backpropagated through the GAE during training. As with AEs and VAEs, GAEs can thus be trained in an *unsupervised manner*, since a loss value can be calculated using only unlabelled instances (a concept which is explored via a worked example in Appendices B.3).

$$\hat{A}_{ij} = \sigma\left(z_i^T z_j\right) \tag{8}$$

5.3 In Summary of Graph Autoencoders

In this section, we have explained the mechanics behind traditional and graph autoencoders. Analagous to how AEs use NN to perform encoding, GAEs use CGNNs to create embeddings. 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 represent a simple method for performing unsupervised training, which allows us to learn powerful embeddings in the graph domain without any labelled data. This concept has not only been used to learn embeddings, but also as the first stage in semi-supervised learning pipelines, where the trained encoder is used to perform downstream machine learning tasks.

5.3.1 Further Reading. Some notable developments in GAEs include variational graph autoencoders (VGAEs) [23] (this architecture is discussed in our worked example in Appendices B.3 and implemented in the associated code), Graph2Gauss (G2G) [2] and Semi-implicit Graph Variational Autoencoders (SIG-VAE) [15], which further apply the concept of embeddings as probability distributions. In addition, both [56] and [33] regularization is achieved through adversarial training. In [16] they take graph embeddings further still: attempting to account for noise in vertex attributes with a denoising attribute autoencoder.

Other applications of AEs to graphs have explored alternative encoders, such as LSTMs, in attempts to more appropriately define a vertex's local neighbourhood and generate embeddings [44, 56]. With a similar perspective to VGAEs, the work described in [2] embeds vertices as Gaussian distributions, rather than point vectors, in order to improve generalisation to unseen vertices and capture uncertainty about each embedding that may be useful to downstream machine learning tasks.

Experiments with alternative decoders have shown promising results in the improvement of GAEs. For example, [15] use a Bernoulli-Poisson link decoder that enables greater generative flexibility and application to sparse real-world graphs. While many GAEs can perform link prediction on undirected graphs, [36] move away from symmetric decoders such as inner-product in order to perform link prediction on directed graphs. This method draws inspiration from physics, particularly classical mechanics, analogising the concepts of mass and acceleration between bodies with edge directions in a graph.

6 CONCLUSION

The development of GNNs has accelerated hugely in the recent years due to increased interest in exploring unstructured data and developing general AI architectures. Consequently GNNs have emerged as a successful and highly performant branch of algorithms for handling such data. Ultimately, GNNs represent both a unique and novel approach to dealing with graphs as input data,

and this capability exposes graph-based datasets to deep learning approaches, thus supporting the development of highly general ML and AI.

ACKNOWLEDGMENTS

We thank our colleague Richard Pienaar for providing feedback which greatly improved this work.

REFERENCES

[1] [n.d.]. Classification: Precision and Recall | Machine Learning Crash Course. https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall

[2] Aleksandar Bojchevski and Stephan Günnemann. 2018. Deep Gaussian Embedding of Graphs: Unsupervised Inductive Learning via Ranking. *arXiv: Machine Learning* (2018).

[3] Xavier Bresson and Thomas Laurent. 2017. Residual Gated Graph ConvNets. arXiv:cs.LG/1711.07553

[4] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. 2017. Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine* 34, 4 (2017), 18–42.

[5] Cameron Buckner and James Garson. 2019. Connectionism. In *The Stanford Encyclopedia of Philosophy* (fall 2019 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

[6] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2016. Deep Neural Networks for Learning Graph Representations. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16)*. AAAI Press, 1145–1152.

[7] Hong Chen and Hisashi Koga. 2019. GL2vec: Graph Embedding Enriched by Line Graphs with Edge Features. In *ICONIP*.

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

[9] 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

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

[11] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[12] Yulan Guo, Hanyun Wang, Qingyong Hu, Hao Liu, Li Liu, and Mohammed Bennamoun. 2019. Deep Learning for 3D Point Clouds: A Survey. arXiv:cs.CV/1912.12033

[13] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. *CoRR* abs/1706.02216 (2017). arXiv:1706.02216 http://arxiv.org/abs/1706.02216

[14] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *CoRR* abs/1709.05584 (2017). arXiv:1709.05584 http://arxiv.org/abs/1709.05584

[15] 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

[16] 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.

[17] G. E. Hinton and R. R. Salakhutdinov. 2006. Reducing the Dimensionality of Data with Neural Networks. *Science* 313, 5786 (2006), 504–507. https://doi.org/10.1126/science.1127647 arXiv:https://science.sciencemag.org/content/313/5786/504.full.pdf

[18] Weihua Hu*, Bowen Liu*, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. 2020. Strategies for Pre-training Graph Neural Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HJlWWJSFDH

[19] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR* abs/1502.03167 (2015). arXiv:1502.03167 http://arxiv.org/abs/1502.03167

[20] 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. *Journal of Chemical Information and Modeling* 52, 7 (2012), 1757–1768. https://doi.org/10.1021/ci3001277 arXiv:https://doi.org/10.1021/ci3001277 PMID: 22587354.

[21] M. A. Khamsi and William A. Kirk. 2001. *An introduction to metric spaces and fixed point theory*. Wiley.

[22] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02907

[23] Thomas N. Kipf and Max Welling. 2016. Variational Graph Auto-Encoders. arXiv:stat.ML/1611.07308

[24] Alex Krizhevsky et al. 2009. Learning multiple layers of features from tiny images. (2009).

[25] Q. V. Le. 2013. Building high-level features using large scale unsupervised learning. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 8595–8598.

[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[27] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. (2010). http://yann.lecun.com/exdb/mnist/

[28] 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 (2017). arXiv:1705.07664 http://arxiv.org/abs/1705.07664

[29] 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. https://doi.org/10.1007/978-1-4939-9761-9_10

[30] Andreas Loukas. 2019. What graph neural networks cannot learn: depth vs width. *CoRR* abs/1907.03199 (2019). arXiv:1907.03199 http://arxiv.org/abs/1907.03199

[31] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning Distributed Representations of Graphs. *CoRR* abs/1707.05005 (2017). arXiv:1707.05005 http://arxiv.org/abs/1707.05005

[32] Antonio Ortega, Pascal Frossard, Jelena Kovačević, José MF Moura, and Pierre Vandergheynst. 2018. Graph signal processing: Overview, challenges, and applications. *Proc. IEEE* 106, 5 (2018), 808–828.

[33] 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

[34] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. 2020. An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs. arXiv:cs.LG/2003.04819

[35] Guillaume Salha, Romain Hennequin, and Michalis Vazirgiannis. 2019. Keep It Simple: Graph Autoencoders Without Graph Convolutional Networks. arXiv:cs.LG/1910.00942

[36] 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 (2019). arXiv:1905.09570 http://arxiv.org/abs/1905.09570

[37] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (Jan 2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[38] Suvash Sedhain, Aditya Krishna Menon, Scott Sanner, and Lexing Xie. 2015. AutoRec: Autoencoders Meet Collaborative Filtering. In *Proceedings of the 24th International Conference on World Wide Web (WWW '15 Companion)*. Association for Computing Machinery, New York, NY, USA, 111–112. https://doi.org/10.1145/2740908.2742726

[39] 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 (2012). arXiv:1211.0053 http://arxiv.org/abs/1211.0053

[40] 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 signal processing magazine* 30, 3 (2013), 83–98.

[41] Tomas Simon, Hanbyul Joo, Iain A. Matthews, and Yaser Sheikh. 2017. Hand Keypoint Detection in Single Images using Multiview Bootstrapping. *CoRR* abs/1704.07809 (2017). arXiv:1704.07809 http://arxiv.org/abs/1704.07809

[42] 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 Signal Processing Magazine* 36, 6 (2019), 133–145.

[43] 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

[44] 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, USA, 2357–2366. https://doi.org/10.1145/3219819.3220068

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

[46] 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 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[47] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. Graph Attention Networks. arXiv:stat.ML/1710.10903

[48] 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. http://papers.nips.cc/paper/6614-hunt-for-the-unique-stable-sparse-and-fast-feature-learning-on-graphs.pdf

[49] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. Extracting and Composing Robust Features with Denoising Autoencoders. In *Proceedings of the 25th International Conference on Machine Learning (ICML '08)*. Association for Computing Machinery, New York, NY, USA, 1096–1103. https://doi.org/10.1145/1390156.1390294

[50] 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, USA, 1225–1234. https://doi.org/10.1145/2939672.2939753

[51] 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. *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019). https://arxiv.org/abs/1909.01315

[52] Isaac Ronald Ward, Hamid Laga, and Mohammed Bennamoun. 2019. RGB-D image-based Object Detection: from Traditional Methods to Deep Learning Techniques. *CoRR* abs/1907.09236 (2019). arXiv:1907.09236 http://arxiv.org/abs/1907.09236

[53] 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 (2019). arXiv:1902.07153 http://arxiv.org/abs/1902.07153

[54] 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 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[55] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks? *CoRR* abs/1810.00826 (2018). arXiv:1810.00826 http://arxiv.org/abs/1810.00826

[56] 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, USA, 2663–2671. https://doi.org/10.1145/3219819.3220000

[57] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. 2019. Graph convolutional networks: a comprehensive review. *Computational Social Networks* 6, 1 (10 Nov 2019), 11. https://doi.org/10.1186/s40649-019-0069-y

[58] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2018. Deep Learning on Graphs: A Survey. *CoRR* abs/1812.04202 (2018). arXiv:1812.04202 http://arxiv.org/abs/1812.04202

[59] 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 (2018). arXiv:1812.08434 http://arxiv.org/abs/1812.08434

[60] Dingyuan Zhu, Peng Cui, Daixin Wang, and Wenwu Zhu. 2018. Deep Variational Network Embedding in Wasserstein Space. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18)*. Association for Computing Machinery, New York, NY, USA, 2827–2836. https://doi.org/10.1145/3219819.3220052

A PERFORMANCE METRICS

In this section we explicitly define the performance metrics used in our practical examples, before presenting the practical examples themselves in Section B. For more detail on these performance metrics, we recommend the reader visit Google's Machine Learning Crash Course [1].

In classification, we define the **positive class** as the class which is relevant or currently being considered, and the **negative class** (or classes) as the remaining class(es). It is customary to label what we are interested in as the positive class. For example, in the case of a machine learning model which predicts if patients have tumors, the positive class would correspond to a patient having a tumor present, and the negative class would correspond to a patient not having a tumor. It then follows that:

- a **true positive** (TP) is where the actual class is **positive** and the prediction is **correct**. For example, the patient has a tumor and the model predicts that they have a tumor.
- a **false positive** (FP) is where the actual class is **negative** and the prediction is **incorrect**. For example, the patient does not have a tumor and the model predicts that they do have a tumor.
- a **true negative** (TN) is where the actual class is **negative** and the prediction is **correct**. For example, the patient does not have a tumor and the model predicts that they do not have a tumor.
- a **false negative** (FN) is where the actual class is **positive** and the prediction is **incorrect**. For example, the patient has a tumor and the model predicts that they do not have a tumor.

Accuracy is the fraction of predictions which were correct (when compared to their ground truth labels) out of all the predictions made.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Number of total predictions}} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{9}$$

Precision is the proportion of correct predictions concerning *only the positive class*, and **recall** is the fraction of *positives* which were correctly predicted.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{10}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{11}$$

The **true positive rate** (TPR) and **false positive rate** (FPR) are defined as follows. Note that the TPR is the same as recall.

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{12}$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{FN}} \tag{13}$$

One can then plot the TPR against the FPR while varying the threshold at which a classification is considered a positive prediction. The resulting plot is the **receiver operating characteristic** (ROC) curve. A sample ROC curve is plotted in Figure 15. The **area under the ROC curve** (AUC) is the integral of the ROC curve in from the bounds 0 to 1 (see Figure 15). It is a useful metric as it is invariant to the classification threshold and scale. In essence, it is an aggregate measure of a model at all classification thresholds.

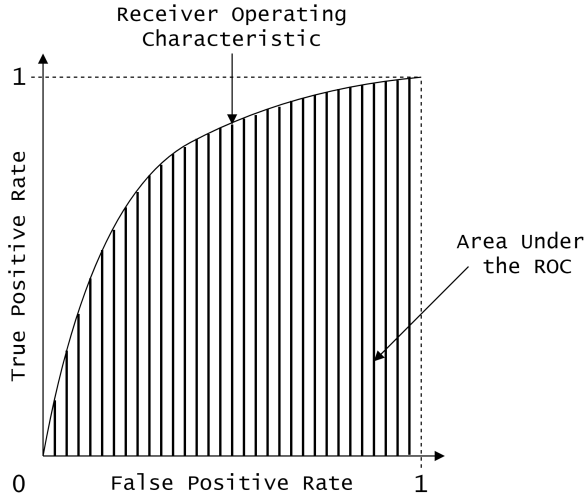


Fig. 15. A demonstration of how the ROC is plotted with respect to the TPR and FPR at varying classification thresholds. The AUC is also illustrated as the shaded area under the receiver operating characteristic.

Average precision is the average value of the precision metric over the interval where recall = 0 to recall = 1. It is also equivalent to measuring the area under the precision-recall curve [52].

$$\text{Average precision} = \int_0^1 \mathrm{p}(r)dr \tag{14}$$

B PRACTICAL EXAMPLES

In the following worked examples, we concisely present the results of an investigation involving GNNs. Code for each of these examples can be accessed online[4].

[4]Code will be made public pending current peer review at https://github.com/isolabs/gnn-tutorial.

B.1 The Role of Recurrent Transitions in RGNNs for Graph Classification

Social networks represent a rich source of graph data, and 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 [34] (available here). GitHub is a code sharing platform with social network elements. *Each* of the 12725 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.

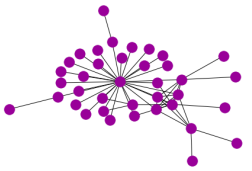


Fig. 16. A web developer group. $N = 38$ and $M = 110$.

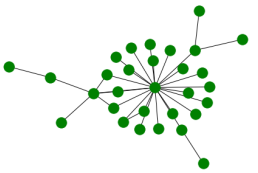


Fig. 17. A machine learning developer group. $N = 30$ and $M = 66$.

Algorithms

Rather than use a true RGNN which 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 which applies a learned set of weights to create 8 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 8096 graphs for 16 epochs and test on 2048 graphs for each architecture.

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 (analagous to more convolutional layers in a CNN).

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

Table 2. The effect of repeated transition function applications on graph classification performance

In fact, we can see that the final hidden representations become more linearly separable (see TSNE visualizations in Figure 18), 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 [7] — 0.551, Graph2Vec [31] — 0.585, SF [9] — 0.558, **FGSD [48] — 0.656** (see Appendices A for a definition of AUC).

Fig. 18. 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.

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 realised through CGNNs in Section 4.1.

B.2 Using GraphSAGE to Generate Embeddings for Unseen Data

The GraphSAGE (**SA**mple and aggre**G**a**t**E) algorithm [13] 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 which 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 19) as provided by the deep learning library *DGL* [51]. The Cora dataset is oft considered 'the MNIST of graph-based learning' and consists of 2708 scientific publications (vertices), each classified into one of seven subfields in AI (or classes). Each vertex has a 1433 element binary feature vector, which indicates if each of the 1433 designated words appeared in the publication.

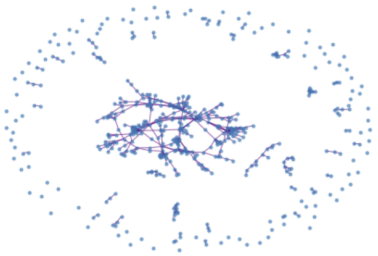


Fig. 19. A subgraph of the Cora dataset. The full Cora graph has $N = 2708$ and $M = 5429$. Note the many vertices with few incident edges (low degree) as compared to the few vertices with many incident edges (high degree).

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 which 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 which can be applied to unseen vertices, rather than reasoning from specific training cases to specific test cases.

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 which 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 which then has its output element-wise max pooled. The layer weights are learned, thus allowing the most import 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 minimise this effect, the order of the input embeddings is randomised. 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**%, **76.3**%, and **74.4**%, respectively. As expected, the learned pool and LSTM aggregators are highly effective, though they incur significant training overheads, and may not be suitable for smaller training graphs or graph datasets. Indeed, in the original GraphSAGE paper [13], 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[a].

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

B.3 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 [11]. In each of these graphs, the vertex features are word vectors indicating the presence or absence of predefined keywords (see Section B.2 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 optimise 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 [23]. 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 in order to optimise 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.

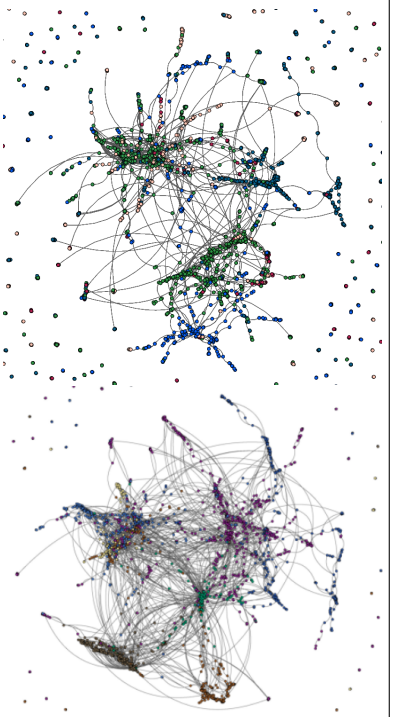


Fig. 20. Renderings of the Citeseer dataset (top) and Cora dataset (bottom). Image best viewed in colour.

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.

Algorithm	Dataset	AUC[a]	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)

Table 3. Comparing the link prediction performance of autoencoder models on Citeseer.

In alignment with [23], 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 ($N = 19717$), and therefore more training examples, than Citesser ($N = 3327$) or Cora ($N = 2708$). Moreover, while Cora and Citeseer vertex features are simple binary word vectors (of sizes 1433 and 3703 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.

[a] For a formal definition of these (and other) performance metrics see Appendices A.