Understanding Convolutions on Graphs

Understanding the building blocks and design choices of graph neural networks.

AUTHORS AFFILIATIONS
Ameya Daigavane Google Research
Balaraman Ravindran Google Research
Gaurav Aggarwal Google Research

PUBLISHED
Sept. 2, 2021

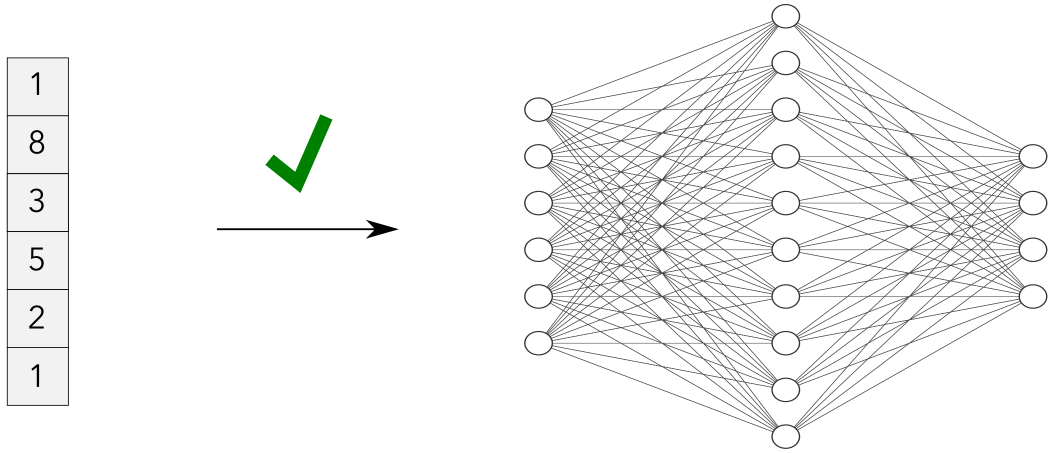
DOI
10.23915/distill.00032

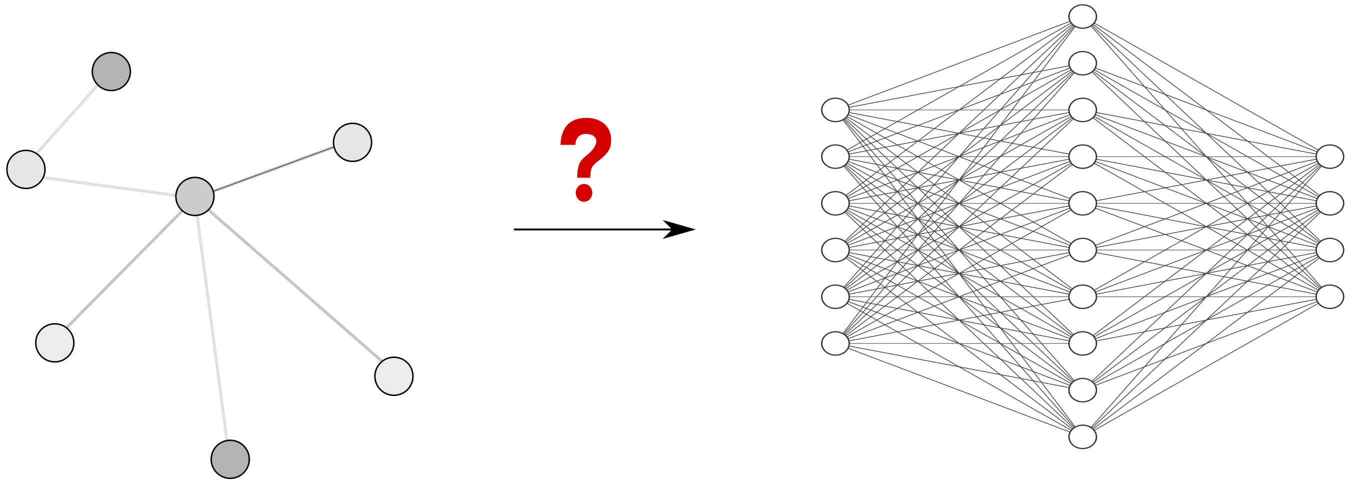
Contents

This article is one of two Distill publications about graph neural networks. Take a look at A Gentle Introduction to Graph Neural Networks for a companion view on many things graph and neural network related.

Many systems and interactions - social networks, molecules, organizations, citations, physical models, transactions - can be represented quite naturally as graphs. How can we reason about and make predictions within these systems?

One idea is to look at tools that have worked well in other domains: neural networks have shown immense predictive power in a variety of learning tasks. However, neural networks have been traditionally used to operate on fixed-size and/or regular-structured inputs (such as sentences, images and video). This makes them unable to elegantly process graph-structured data.





Graph neural networks (GNNs) are a family of neural networks that can operate naturally on graph-structured data. By extracting and utilizing features from the underlying graph, GNNs can make more informed predictions about entities in these interactions, as compared to models that consider individual entities in isolation.

GNNs are not the only tools available to model graph-structured data: graph kernels and random-walk methods were some of the most popular ones. Today, however, GNNs have largely replaced these techniques because of their inherent flexibility to model the underlying systems better.

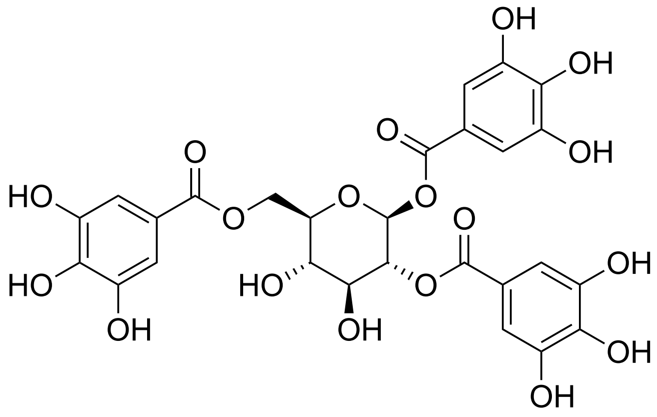
In this article, we will illustrate the challenges of computing over graphs, describe the origin and design of graph neural networks, and explore the most popular GNN variants in recent times. Particularly, we will see that many of these variants are composed of similar building blocks.

First, let's discuss some of the complications that graphs come with.

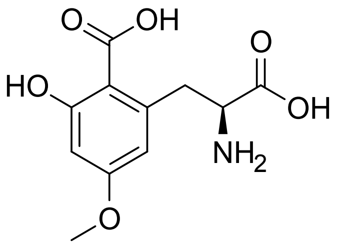
The Challenges of Computation on Graphs

Lack of Consistent Structure

Graphs are extremely flexible mathematical models; but this means they lack consistent structure across instances. Consider the task of predicting whether a given chemical molecule is toxic :



Left: A non-toxic 1,2,6-trigalloyl-glucose molecule.



Right: A toxic caramboxin molecule.

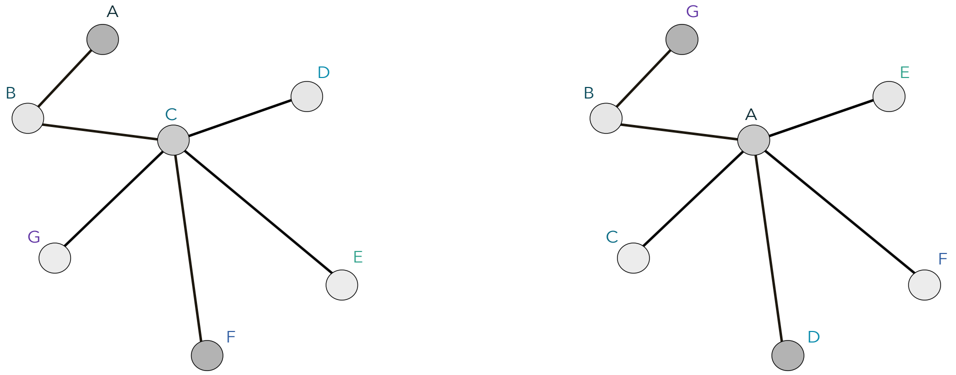
Looking at a few examples, the following issues quickly become apparent:

- Molecules may have different numbers of atoms.
- The atoms in a molecule may be of different types.
- Each of these atoms may have different number of connections.
- These connections can have different strengths.

Representing graphs in a format that can be computed over is non-trivial, and the final representation chosen often depends significantly on the actual problem.

Node-Order Equivariance

Extending the point above: graphs often have no inherent ordering present amongst the nodes. Compare this to images, where every pixel is uniquely determined by its absolute position within the image!



Representing the graph as one vector requires us to fix an order on the nodes. But what do we do when the nodes have no inherent order? **Above:** The same graph labelled in two different ways. The alphabets indicate the ordering of the nodes.

As a result, we would like our algorithms to be node-order equivariant: they should not depend on the ordering of the nodes of the graph. If we permute the nodes in some way, the resulting representations of the nodes as computed by our algorithms should also be permuted in the same way.

Scalability

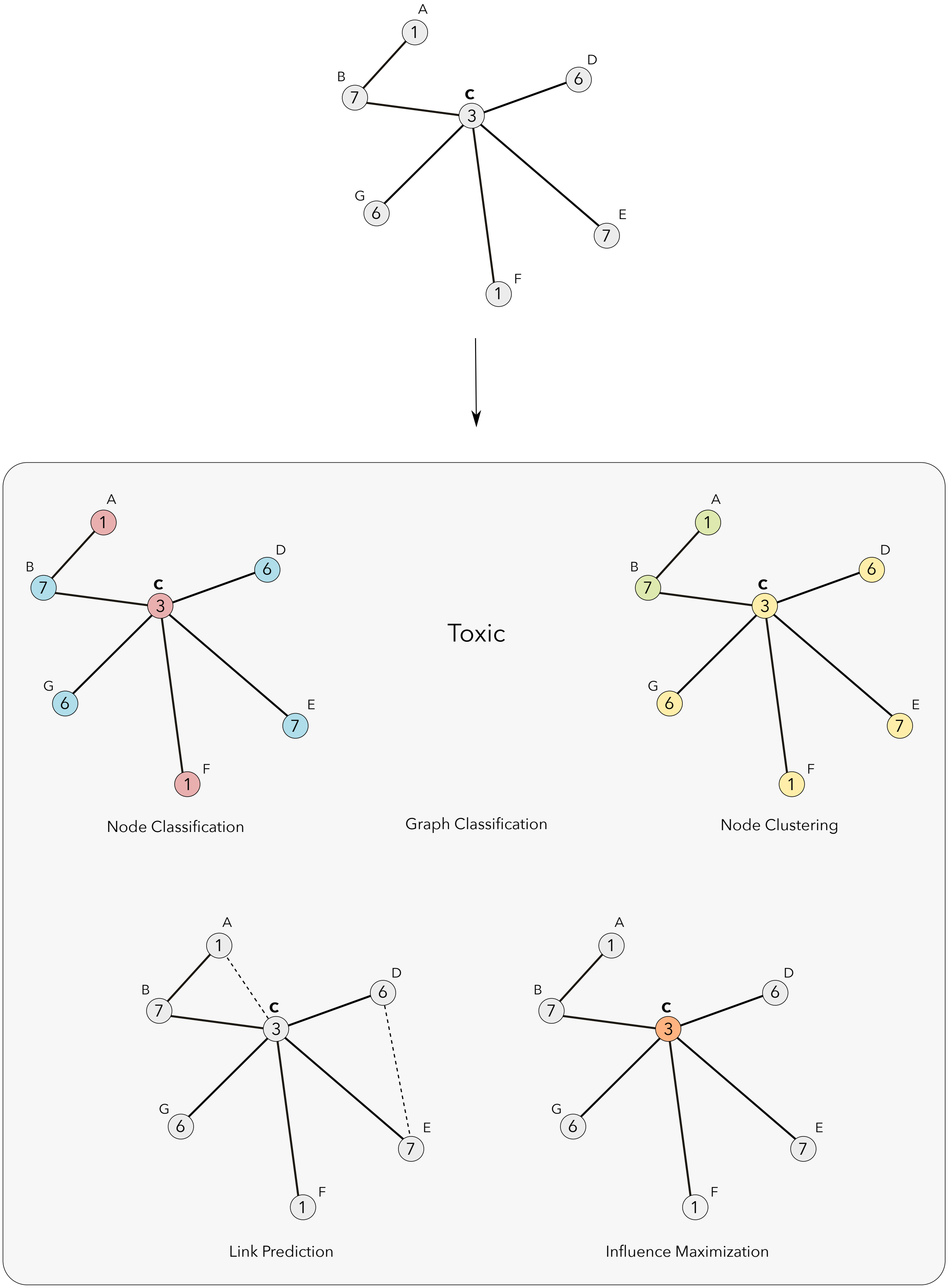
Graphs can be really large! Think about social networks like Facebook and Twitter, which have over a billion users. Operating on data this large is not easy.

Luckily, most naturally occuring graphs are ‘sparse’: they tend to have their number of edges linear in their number of vertices. We will see that this allows the use of clever methods to efficiently compute representations of nodes within the graph. Further, the methods that we look at here will have significantly fewer parameters in comparison to the size of the graphs they operate on.

Problem Setting and Notation

There are many useful problems that can be formulated over graphs:

- **Node Classification:** Classifying individual nodes.
- **Graph Classification:** Classifying entire graphs.
- **Node Clustering:** Grouping together similar nodes based on connectivity.
- **Link Prediction:** Predicting missing links.
- **Influence Maximization:** Identifying influential nodes.



Examples of problems that can be defined over graphs. This list is not exhaustive!

A common precursor in solving many of these problems is **node representation learning**: learning to map individual nodes to fixed-size real-valued vectors (called 'representations' or 'embeddings').

In Learning GNN Parameters, we will see how the learnt embeddings can be used for these tasks.

Different GNN variants are distinguished by the way these representations are computed. Generally, however, GNNs compute node representations in an iterative process. We will use the notation $h_v^{(k)}$ to indicate the representation of node v after the k^{th} iteration. Each iteration can be thought of as the equivalent of a 'layer' in standard neural networks.

We will define a graph G as a set of nodes, V, with a set of edges E connecting them. Nodes can have individual features as part of the input: we will denote by x_v the individual feature for node $v \in V$. For example, the 'node features' for a pixel in a color image would be the red, green and blue channel (RGB) values at that pixel.

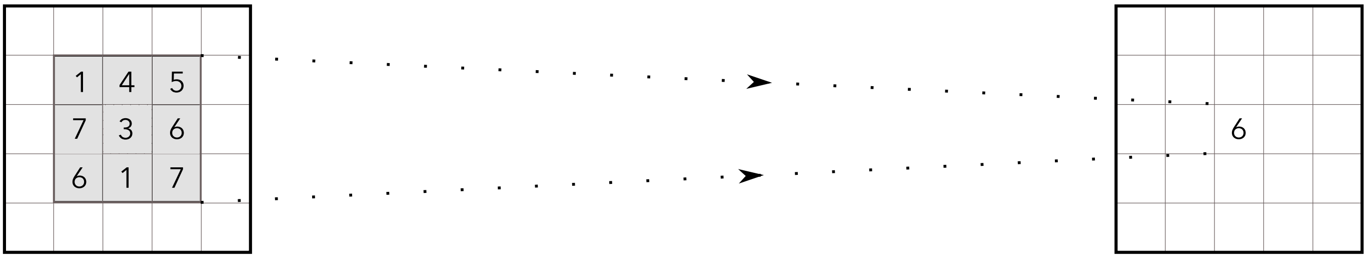
For ease of exposition, we will assume G is undirected, and all nodes are of the same type. These kinds of graphs are called 'homogeneous'. Many of the same ideas we will see here apply to other kinds of graphs: we will discuss this later in Different Kinds of Graphs.

Sometimes we will need to denote a graph property by a matrix M, where each row M_v represents a property corresponding to a particular vertex v.

Extending Convolutions to Graphs

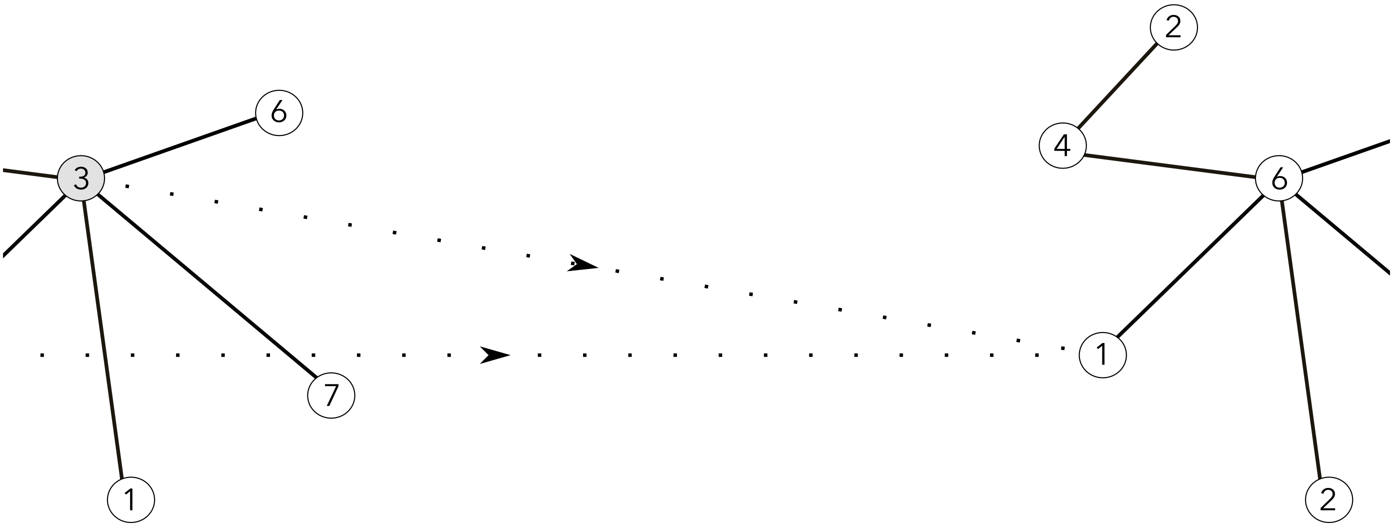
Convolutional Neural Networks have been seen to be quite powerful in extracting features from images. However, images themselves can be seen as graphs with a very regular grid-like structure, where the individual pixels are nodes, and the RGB channel values at each pixel as the node features.

A natural idea, then, is to consider generalizing convolutions to arbitrary graphs. Recall, however, the challenges listed out in the previous section: in particular, ordinary convolutions are not node-order invariant, because they depend on the absolute positions of pixels. It is initially unclear as how to generalize convolutions over grids to convolutions over general graphs, where the neighbourhood structure differs from node to node. The curious reader may wonder if performing some sort of padding and ordering could be done to ensure the consistency of neighbourhood structure across nodes. This has been attempted with some success , but the techniques we will look at here are more general and powerful.



Convolution in CNNs

Convolutions in CNNs are inherently localized. Neighbours participating in the convolution at the center pixel are highlighted in gray.



Localized Convolution in GNNs

GNNs can perform localized convolutions mimicking CNNs. Hover over a node to see its immediate neighbourhood highlighted on the left. The structure of this neighbourhood changes from node to node.

We begin by introducing the idea of constructing polynomial filters over node neighbourhoods, much like how CNNs compute localized filters over neighbouring pixels. Then, we will see how more recent approaches extend on this idea with more powerful mechanisms. Finally, we will discuss alternative methods that can use 'global' graph-level information for computing node representations.

Polynomial Filters on Graphs

The Graph Laplacian

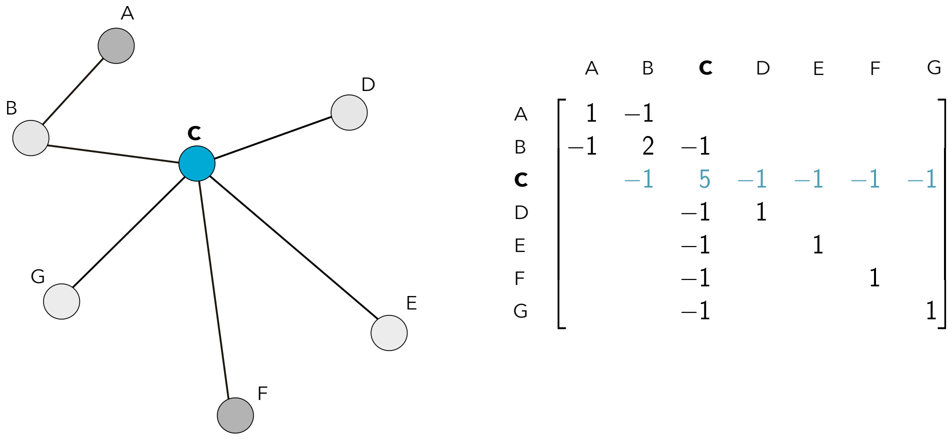
Given a graph G, let us fix an arbitrary ordering of the n nodes of G. We denote the $0-1$ adjacency matrix of G by A, we can construct the diagonal degree matrix D of G as:

$$D_v = \sum_u A_{vu}.$$

The degree of node v is the number of edges incident at v.

where A_{vu} denotes the entry in the row corresponding to v and the column corresponding to u in the matrix A. We will use this notation throughout this section.

Then, the graph Laplacian L is the square $n \times n$ matrix defined as: $L = D - A$.



$$\begin{array}{c|ccccccc} & A & B & \mathbf{C} & D & E & F & G \\ \hline A & 1 & -1 & & & & & \\ B & -1 & 2 & -1 & & & & \\ \mathbf{C} & & -1 & 5 & -1 & -1 & -1 & -1 \\ D & & & -1 & 1 & & & \\ E & & & -1 & & 1 & & \\ F & & & -1 & & & 1 & \\ G & & & -1 & & & & 1 \end{array}$$

Input Graph G

Laplacian L of G

The Laplacian L for an undirected graph G, with the row corresponding to node **C** highlighted. Zeros in L are not displayed above. The Laplacian L depends only on the structure of the graph G, not on any node features.

The graph Laplacian gets its name from being the discrete analog of the Laplacian operator from calculus.

Although it encodes precisely the same information as the adjacency matrix A In the sense that given either of the matrices A or L, you can construct the other. , the graph Laplacian has many interesting properties of its own. The graph Laplacian shows up in many mathematical problems involving graphs: random walks, spectral clustering, and diffusion, to name a few. We will see some of these properties in a later section, but will instead point readers to this tutorial for greater insight into the graph Laplacian.

Polynomials of the Laplacian

Now that we have understood what the graph Laplacian is, we can build polynomials of the form:

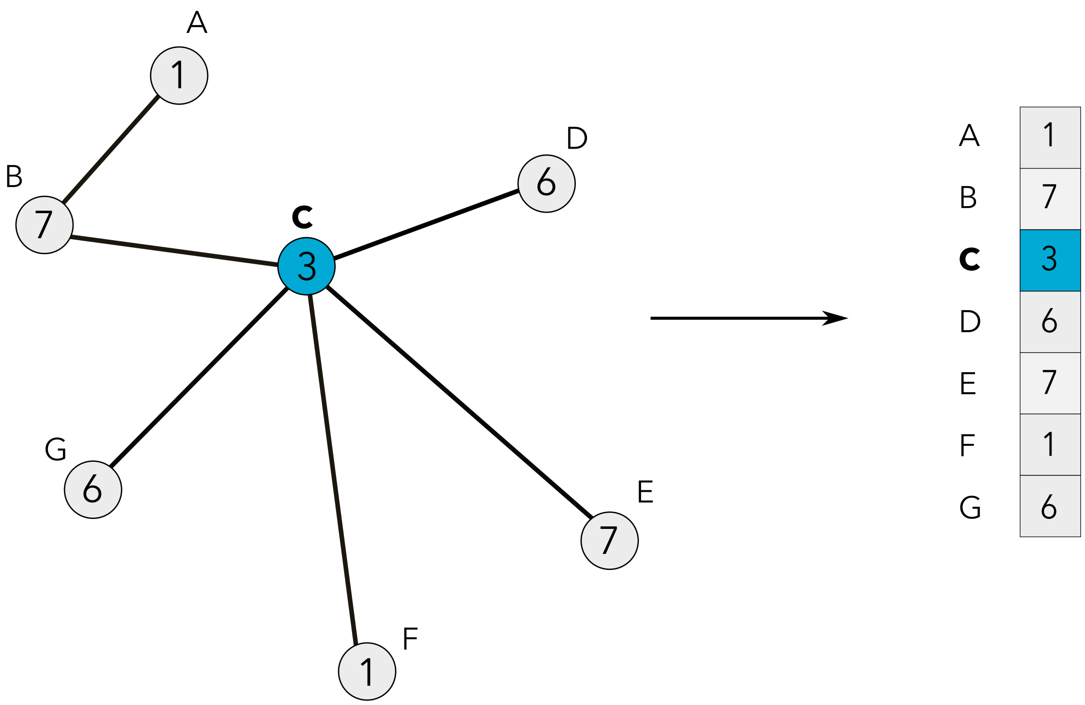
$$p_w(L) = w_0 I_n + w_1 L + w_2 L^2 + \ldots + w_d L^d = \sum_{i=0}^{d} w_i L^i.$$

Each polynomial of this form can alternately be represented by its vector of coefficients $w = [w_0, \ldots, w_d]$. Note that for every w, $p_w(L)$ is an $n \times n$ matrix, just like L.

These polynomials can be thought of as the equivalent of 'filters' in CNNs, and the coefficients w as the weights of the 'filters'.

For ease of exposition, we will focus on the case where nodes have one-dimensional features: each of the x_v for $v \in V$ is just a real number. The same ideas hold when each of the x_v are higher-dimensional vectors, as well.

Using the previously chosen ordering of the nodes, we can stack all of the node features x_v to get a vector $x \in \mathbb{R}^n$.



Fixing a node order (indicated by the alphabets) and collecting all node features into a single vector x.

Once we have constructed the feature vector x, we can define its convolution with a polynomial filter p_w as:

$$x' = p_w(L)\ x$$

To understand how the coefficients w affect the convolution, let us begin by considering the 'simplest' polynomial: when $w_0 = 1$ and all of the other coefficients are 0. In this case, x' is just x:

$$x' = p_w(L)\ x = \sum_{i=0}^{d} w_i L^i x = w_0 I_n x = x.$$

Now, if we increase the degree, and consider the case where instead $w_1 = 1$ and and all of the other coefficients are 0. Then, x' is just Lx, and so:

$$\begin{aligned} x'_v = (Lx)_v &= L_v x \\ &= \sum_{u \in G} L_{vu} x_u \\ &= \sum_{u \in G} (D_{vu} - A_{vu}) x_u \\ &= D_v\ x_v - \sum_{u \in \mathcal{N}(v)} x_u \end{aligned}$$

We see that the features at each node v are combined with the features of its immediate neighbours $u \in \mathcal{N}(v)$. For readers familiar with Laplacian filtering of images, this is the exact same idea. When x is an image, $x' = Lx$ is exactly the result of applying a 'Laplacian filter' to x.

At this point, a natural question to ask is: How does the degree d of the polynomial influence the behaviour of the convolution? Indeed, it is not too hard to show that: This is Lemma 5.2 from .

$$\text{dist}_G(v, u) > i \quad \Longrightarrow \quad L_{vu}^i = 0.$$

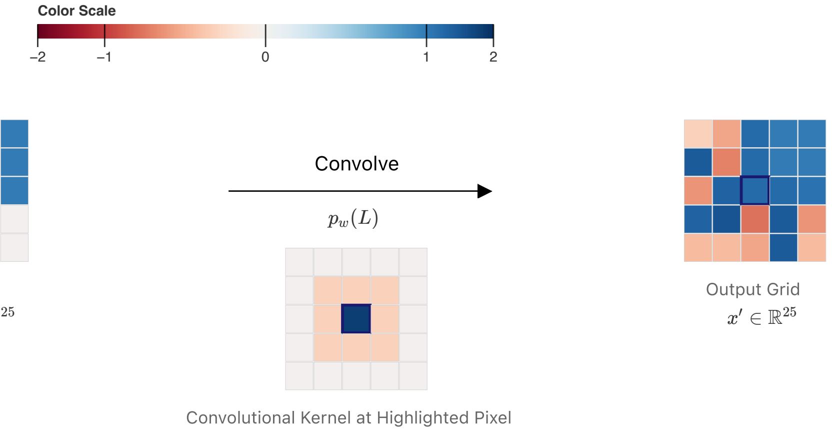
This implies, when we convolve x with $p_w(L)$ of degree d to get x':

$$\begin{aligned} x'_v = (p_w(L)x)_v &= (p_w(L))_v x \\ &= \sum_{i=0}^{d} w_i L_v^i x \\ &= \sum_{i=0}^{d} w_i \sum_{u \in G} L_{vu}^i x_u \\ &= \sum_{i=0}^{d} w_i \sum_{\substack{u \in G \\ \text{dist}_G(v,u) \leq i}} L_{vu}^i x_u. \end{aligned}$$

Effectively, the convolution at node v occurs only with nodes u which are not more than d hops away. Thus, these polynomial filters are localized. The degree of the localization is governed completely by d.

To help you understand these 'polynomial-based' convolutions better, we have created the visualization below. Vary the polynomial coefficients and the input grid x to see how the result x' of the convolution changes. The grid under the arrow shows the equivalent convolutional kernel applied at the highlighted pixel in x to get the resulting pixel in x'. The kernel corresponds to the row of $p_w(L)$ for the highlighted pixel. Note that even after adjusting for position, this kernel is different for different pixels, depending on their position within the grid.

Reset Grid | Randomize Grid



$$p_w(L) = \sum_{i=0}^{2} w_i L^i = 1I \; + \; 0.1L \; + \; 0L^2.$$

w_0	w_1	w_2
1	0.1	0

Choice of Laplacian

● Unnormalized L ○ Normalized $\tilde{L}$

Reset Coefficients

Hover over a pixel in the input grid (left, representing x) to highlight it and see the equivalent convolutional kernel for that pixel under the arrow. The result x' of the convolution is shown on the right: note that different convolutional kernels are applied at different pixels, depending on their location.

Click on the input grid to toggle pixel values between 0 (white) and 1 (blue). To randomize the input grid, press 'Randomize Grid'. To reset all pixels to 0, press 'Reset Grid'. Use the sliders at the bottom to change the coefficients w. To reset all coefficients w to 0, press 'Reset Coefficients.'

ChebNet

ChebNet refines this idea of polynomial filters by looking at polynomial filters of the form:

$$p_w(L) = \sum_{i=1}^{d} w_i T_i(\tilde{L})$$

where T_i is the degree-i Chebyshev polynomial of the first kind and $\tilde{L}$ is the normalized Laplacian defined using the largest eigenvalue of L: We discuss the eigenvalues of the Laplacian L in more detail in a later section.

$$\tilde{L} = \frac{2L}{\lambda_{\max}(L)} - I_n.$$

What is the motivation behind these choices?

- L is actually positive semi-definite: all of the eigenvalues of L are not lesser than 0. If $\lambda_{\max}(L) > 1$, the entries in the powers of L rapidly increase in size. $\tilde{L}$ is effectively a scaled-down version of L, with eigenvalues guaranteed to be in the range $[-1, 1]$. This prevents the entries of powers of $\tilde{L}$ from blowing up. Indeed, in the visualization above: we restrict the higher-order coefficients when the unnormalized Laplacian L is selected, but allow larger values when the normalized Laplacian $\tilde{L}$ is selected, in order to show the result x' on the same color scale.
- The Chebyshev polynomials have certain interesting properties that make interpolation more numerically stable. We won't talk about this in more depth here, but will advise interested readers to take a look at as a definitive resource.

Polynomial Filters are Node-Order Equivariant

The polynomial filters we considered here are actually independent of the ordering of the nodes. This is particularly easy to see when the degree of the polynomial p_w is 1: where each node's feature is aggregated with the sum of its neighbour's features. Clearly, this sum does not depend on the order of the neighbours. A similar proof follows for higher degree polynomials: the entries in the powers of L are equivariant to the ordering of the nodes.

Details for the Interested Reader
As above, let's assume an arbitrary node-order over the n nodes of our graph. Any other node-order can be thought of as a permutation of this original node-order. We can represent any permutation by a permutation matrix P. P will always be an orthogonal $0-1$ matrix:

$$PP^T = P^TP = I_n.$$

Then, we call a function f node-order equivariant iff for all permutations P:

$$f(Px) = Pf(x).$$

When switching to the new node-order using the permutation P, the quantities below transform in the following way:

$$\begin{aligned} x &\to Px \\ L &\to PLP^T \\ L^i &\to PL^iP^T \end{aligned}$$

and so, for the case of polynomial filters where $f(x) = p_w(L)\ x$, we can see that:

$$\begin{aligned} f(Px) &= \sum_{i=0}^{d} w_i (PL^iP^T)(Px) \\ &= P \sum_{i=0}^{d} w_i L^i x \\ &= Pf(x). \end{aligned}$$

as claimed.

Embedding Computation

We now describe how we can build a graph neural network by stacking ChebNet (or any polynomial filter) layers one after the other with nonlinearities, much like a standard CNN. In particular, if we have K different polynomial filter layers, the k^{th} of which has its own learnable weights $w^{(k)}$, we would perform the following computation:

Start with the original features.

$$h^{(0)} = x$$

Color Codes:

□ Computed node embeddings.

□ Learnable parameters.

Then iterate, for $k = 1, 2, \ldots$ upto K:

$$p^{(k)} = p_{w^{(k)}}(L)$$

Compute the matrix $p^{(k)}$ as the polynomial defined by the filter weights $w^{(k)}$ evaluated at L.

$$g^{(k)} = p^{(k)} \times h^{(k-1)}$$

Multiply $p^{(k)}$ with $h^{(k-1)}$: a standard matrix-vector multiply operation.

$$h^{(k)} = \sigma\left(g^{(k)}\right)$$

Apply a non-linearity σ to $g^{(k)}$ to get $h^{(k)}$.

Note that these networks reuse the same filter weights across different nodes, exactly mimicking weight-sharing in Convolutional Neural Networks (CNNs) which reuse weights for convolutional filters across a grid.

Modern Graph Neural Networks

ChebNet was a breakthrough in learning localized filters over graphs, and it motivated many to think of graph convolutions from a different perspective.

We return back to the result of convolving x by the polynomial kernel $p_w(L) = L$, focussing on a particular vertex v:

$$\begin{aligned}(Lx)_v &= L_v x \\ &= \sum_{u \in G} L_{vu} x_u \\ &= \sum_{u \in G} (D_{vu} - A_{vu}) x_u \\ &= D_v \, x_v - \sum_{u \in \mathcal{N}(v)} x_u\end{aligned}$$

As we noted before, this is a 1-hop localized convolution. But more importantly, we can think of this convolution as arising of two steps:

- Aggregating over immediate neighbour features x_u.
- Combining with the node's own feature x_v.

Key Idea: What if we consider different kinds of 'aggregation' and 'combination' steps, beyond what are possible using polynomial filters?

By ensuring that the aggregation is node-order equivariant, the overall convolution becomes node-order equivariant.

These convolutions can be thought of as 'message-passing' between adjacent nodes: after each step, every node receives some 'information' from its neighbours.

By iteratively repeating the 1-hop localized convolutions K times (i.e., repeatedly 'passing messages'), the receptive field of the convolution effectively includes all nodes upto K hops away.

Embedding Computation

Message-passing forms the backbone of many GNN architectures today. We describe the most popular ones in depth below:

- Graph Convolutional Networks (GCN)
- Graph Attention Networks (GAT)
- Graph Sample and Aggregate (GraphSAGE)

- Graph Isomorphism Network (GIN)

GCN GAT GraphSAGE GIN

$$h_v^{(0)} \quad = \quad x_v \quad \text{for all } v \in V.$$

Node v's initial embedding.

... is just node v's original features.

and for $k = 1, 2, \dots$ upto K:

$$h_v^{(k)} \quad = \quad f^{(k)}\left(W^{(k)} \cdot \frac{\sum\limits_{u \in \mathcal{N}(v)} h_u^{(k-1)}}{|\mathcal{N}(v)|} + B^{(k)} \cdot h_v^{(k-1)}\right) \quad \text{for all } v \in V.$$

Node v's embedding at step k.

Mean of v's neighbour's embeddings at step $k-1$.

Node v's embedding at step $k-1$.

Color Codes:

□ Embedding of node v.

□ Embedding of a neighbour of node v.

□ (Potentially) Learnable parameters.

Predictions can be made at each node by using the final computed embedding:

$$\hat{y_v} = \text{PREDICT}(h_v^{(K)})$$

where PREDICT is generally another neural network, learnt together with the GCN model.

For each step k, the function $f^{(k)}$, matrices $W^{(k)}$ and $B^{(k)}$ are shared across all nodes.

This allows the GCN model to scale well, because the number of parameters in the model is not tied to the size of the graph.

The variant we discuss here is the 2-parameter model from the original paper , which is more expressive. We also consider the following normalization (iteration subscripts omitted):

$$f\left(W \cdot \sum_{u \in \mathcal{N}(v)} \frac{h_u}{|\mathcal{N}(v)|} + B \cdot h_v\right)$$

instead of the normalization defined in the original paper:

$$f\left(W \cdot \sum_{u \in \mathcal{N}(v)} \frac{h_u}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}} + B \cdot h_v\right)$$

for ease of exposition.

Thoughts

An interesting point is to assess different aggregation functions: are some better and others worse? demonstrates that aggregation functions indeed can be compared on how well they can uniquely preserve node neighbourhood features; we recommend the interested reader take a look at the detailed theoretical analysis there.

Here, we've talk about GNNs where the computation only occurs at the nodes. More recent GNN models such as Message-Passing Neural Networks and Graph Networks perform computation over the edges as well; they compute edge embeddings together with node embeddings. This is an even more general framework - but the same 'message passing' ideas from this section apply.

Interactive Graph Neural Networks

Below is an interactive visualization of these GNN models on small graphs. For clarity, the node features are just real numbers here, shown inside the squares next to each node, but the same equations hold when the node features are vectors.

GCN GAT GraphSAGE GIN

Reset Undo Last Update Update All Nodes Randomize Graph

Initial Graph

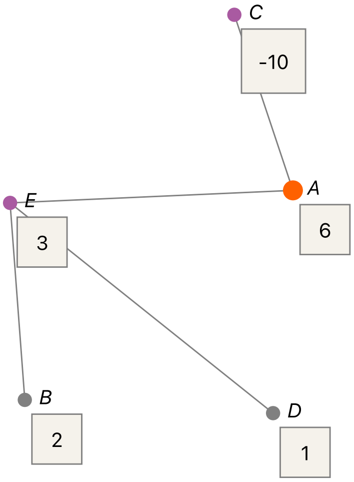
Parameters for Next Update

$W^{(1)}$

1

$B^{(1)}$

1



Next Update (Iteration 1):

Equation for Node A:

$$
\begin{aligned}
h_A^{(1)} &= f\left(W^{(1)} \times \frac{h_C^{(0)} + h_E^{(0)}}{2} + B^{(1)} \times h_A^{(0)}\right) \\
&= f\left(1 \times \frac{-10+3}{2} + 1 \times 6\right) \\
&= f\left(-3.5 + 6\right) \\
&= f\left(2.5\right) \\
&= \mathrm{ReLU}\left(2.5\right) = 2.5.
\end{aligned}
$$

Here, f is just ReLU: $f(x) = \max(x, 0)$.

Note that the weights $W^{(1)}$ and $B^{(1)}$ are shared across all nodes!

Choose a GNN model using the tabs at the top. Click on a node to see the update equation at that node for the next iteration. Use the sliders on the left to change the weights for the current iteration, and watch how the update equation changes.

In practice, each iteration above is generally thought of as a single ‘neural network layer’. This ideology is followed by many popular Graph Neural Network libraries, For example: PyTorch Geometric and StellarGraph. allowing one to compose different types of graph convolutions in the same model.

From Local to Global Convolutions

The methods we've seen so far perform 'local' convolutions: every node's feature is updated using a function of its local neighbours' features.

While performing enough steps of message-passing will eventually ensure that information from all nodes in the graph is passed, one may wonder if there are more direct ways to perform 'global' convolutions.

The answer is yes; we will now describe an approach that was actually first put forward in the context of neural networks by , much before any of the GNN models we looked at above.

Spectral Convolutions

As before, we will focus on the case where nodes have one-dimensional features. After choosing an arbitrary node-order, we can stack all of the node features to get a 'feature vector' $x \in \mathbb{R}^n$.

Key Idea: Given a feature vector x, the Laplacian L allows us to quantify how smooth x is, with respect to G.

How?

After normalizing x such that $\sum_{i=1}^{n} x_i^2 = 1$, if we look at the following quantity involving L: R_L is formally called the Rayleigh quotient.

$$R_L(x) = \frac{x^T L x}{x^T x} = \frac{\sum_{(i,j) \in E} (x_i - x_j)^2}{\sum_i x_i^2} = \sum_{(i,j) \in E} (x_i - x_j)^2.$$

we immediately see that feature vectors x that assign similar values to adjacent nodes in G (hence, are smooth) would have smaller values of $R_L(x)$.

L is a real, symmetric matrix, which means it has all real eigenvalues $\lambda_1 \leq \ldots \leq \lambda_n$. An eigenvalue λ of a matrix A is a value satisfying the equation $Au = \lambda u$ for a certain vector u, called an eigenvector. For a friendly introduction to eigenvectors, please see this tutorial. Further, the corresponding eigenvectors $u_1, \ldots, u_n$ can be taken to be orthonormal:

$$u_{k_1}^T u_{k_2} = \begin{cases} 1 & \text{if } k_1 = k_2. \\ 0 & \text{if } k_1 \neq k_2. \end{cases}$$

It turns out that these eigenvectors of L are successively less smooth, as R_L indicates: This is the min-max theorem for eigenvalues.

$$\underset{x,\ x \perp \{u_1, \ldots, u_{i-1}\}}{\operatorname{argmin}} R_L(x) = u_i. \qquad \min_{x,\ x \perp \{u_1, \ldots, u_{i-1}\}} R_L(x) = \lambda_i.$$

The set of eigenvalues of L are called its 'spectrum', hence the name! We denote the 'spectral' decomposition of L as:

$$L = U \Lambda U^T.$$

where Λ is the diagonal matrix of sorted eigenvalues, and U denotes the matrix of the eigenvectors (sorted corresponding to increasing eigenvalues):

$$\Lambda = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \qquad U = \begin{bmatrix} u_1 & \cdots & u_n \end{bmatrix}.$$

The orthonormality condition between eigenvectors gives us that $U^T U = I$, the identity matrix. As these n eigenvectors form a basis for $\mathbb{R}^n$, any feature vector x can be represented as a linear combination of these eigenvectors:

$$x = \sum_{i=1}^{n} \hat{x_i} u_i = U \hat{x}.$$

where $\hat{x}$ is the vector of coefficients $[x_0, \ldots x_n]$. We call $\hat{x}$ as the spectral representation of the feature vector x. The orthonormality condition allows us to state:

$$x = U\hat{x} \quad \Longleftrightarrow \quad U^T x = \hat{x}.$$

This pair of equations allows us to interconvert between the 'natural' representation x and the 'spectral' representation $\hat{x}$ for any vector $x \in \mathbb{R}^n$.

Spectral Representations of Natural Images

As discussed before, we can consider any image as a grid graph, where each pixel is a node, connected by edges to adjacent pixels. Thus, a pixel can have either $3, 5,$ or 8 neighbours, depending on its location within the image grid. Each pixel gets a value as part of the image. If the image is grayscale, each value will be a single real number indicating how dark the pixel is. If the image is colored, each value will be a 3-dimensional vector, indicating the values for the red, green and blue (RGB) channels. We use the alpha channel as well in the visualization below, so this is actually RGBA.

This construction allows us to compute the graph Laplacian and the eigenvector matrix U. Given an image, we can then investigate what its spectral representation looks like.

To shed some light on what the spectral representation actually encodes, we perform the following experiment over each channel of the image independently:

- We first collect all pixel values across a channel into a feature vector x.
- Then, we obtain its spectral representation $\hat{x}$.

$$\hat{x} = U^T x$$

- We truncate this to the first m components to get $\hat{x}_m$. By truncation, we mean zeroing out all of the remaining $n - m$ components of $\hat{x}$. This truncation is equivalent to using only the first m eigenvectors to compute the spectral representation.

$$\hat{x}_m = \text{Truncate}_m(\hat{x})$$

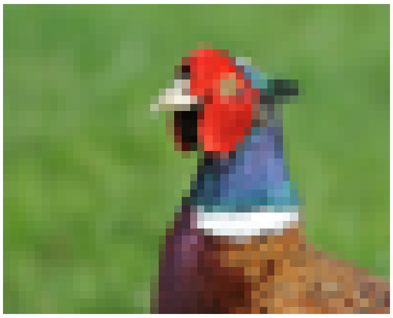
- Then, we convert this truncated representation $\hat{x}_m$ back to the natural basis to get x_m.

$$x_m = U\hat{x}_m$$

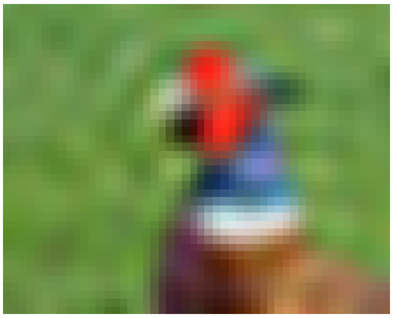
Finally, we stack the resulting channels back together to get back an image. We can now see how the resulting image changes with choices of m. Note that when $m = n$, the resulting image is identical to the original image, as we can reconstruct each channel exactly.

Sample Image

● Chicken ○ Fish ○ Frog ○ Spider



Keep First 200 Spectral Components



Original Image x

Transformed Image x'

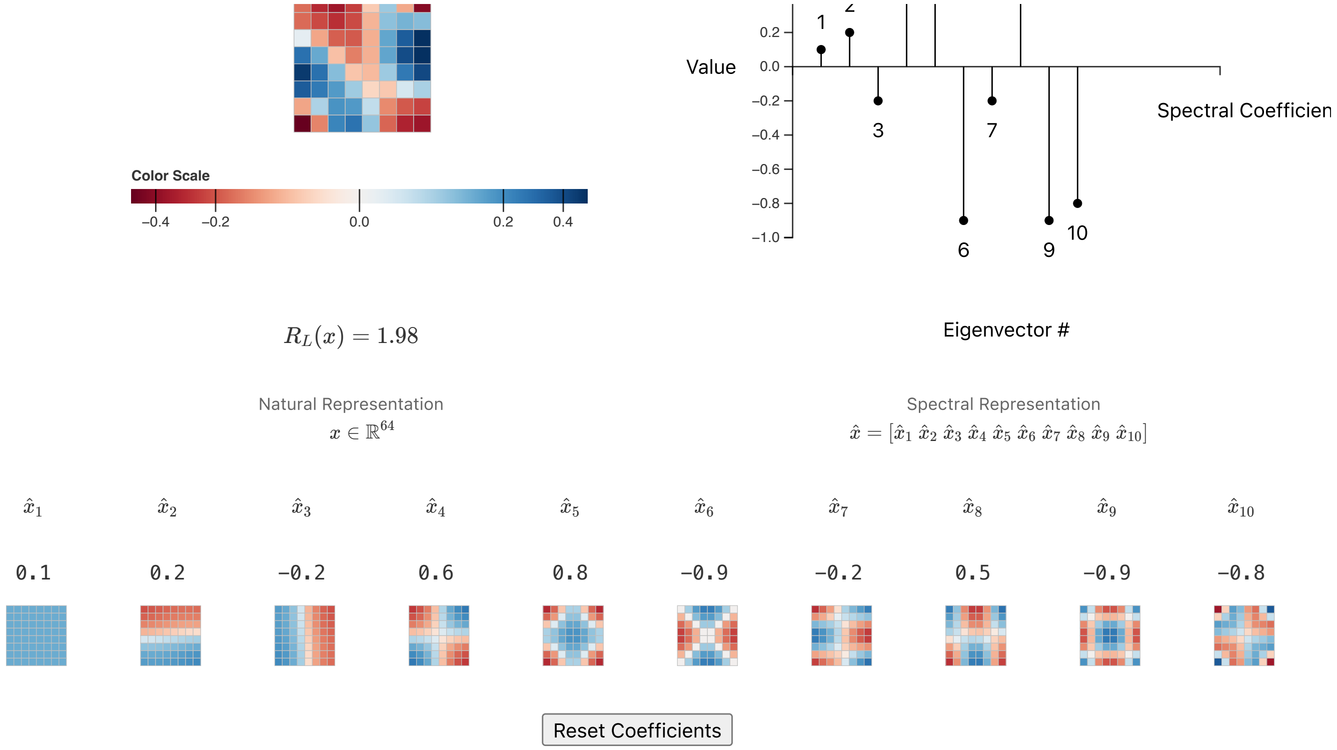
Number of Spectral Components (m)

200

Use the radio buttons at the top to chose one of the four sample images. Each of these images has been taken from the ImageNet dataset and downsampled to 50 pixels wide and 40 pixels tall. As there are $n = 50 \times 40 = 2000$ pixels in each image, there are 2000 Laplacian eigenvectors. Use the slider at the bottom to change the number of spectral components to keep, noting how images get progressively blurrier as the number of components decrease.

As m decreases, we see that the output image x_m gets blurrier. If we decrease m to 1, the output image x_m is entirely the same color throughout. We see that we do not need to keep all n components; we can retain a lot of the information in the image with significantly fewer components. We can relate this to the Fourier decomposition of images: the more eigenvectors we use, the higher frequencies we can represent on the grid.

To complement the visualization above, we additionally visualize the first few eigenvectors on a smaller 8×8 grid below. We change the coefficients of the first 10 out of 64 eigenvectors in the spectral representation and see how the resulting image changes:



Move the sliders to change the spectral representation $\hat{x}$ (right), and see how x itself changes on the image (left). Note how the first eigenvectors are much 'smoother' than the later ones, and the many patterns we can make with only 10 eigenvectors.

These visualizations should convince you that the first eigenvectors are indeed smooth, and the smoothness correspondingly decreases as we consider later eigenvectors.

For any image x, we can think of the initial entries of the spectral representation $\hat{x}$ as capturing 'global' image-wide trends, which are the low-frequency components, while the later entries as capturing 'local' details, which are the high-frequency components.

Embedding Computation

We now have the background to understand spectral convolutions and how they can be used to compute embeddings/feature representations of nodes.

As before, the model we describe below has K layers: each layer k has learnable parameters $\hat{w}^{(k)}$, called the 'filter weights'. These weights will be convolved with the spectral representations of the node features. As a result, the number of weights needed in each layer is equal to m, the number of eigenvectors used to compute the spectral representations. We had shown in the previous section that we can take $m \ll n$ and still not lose out on significant amounts of information.

Thus, convolution in the spectral domain enables the use of significantly fewer parameters than just direct convolution in the natural domain. Further, by virtue of the smoothness of the Laplacian eigenvectors across the graph, using spectral representations automatically enforces an inductive bias for neighbouring nodes to get similar representations.

Assuming one-dimensional node features for now, the output of each layer is a vector of node representations $h^{(k)}$, where each node's representation corresponds to a row of the vector.

$$h^{(k)} = \begin{bmatrix} h_1^{(k)} \\ \vdots \\ h_n^{(k)} \end{bmatrix} \qquad \text{for each } k = 0, 1, 2, \ldots \text{ upto } K.$$

We fix an ordering of the nodes in G. This gives us the adjacency matrix A and the graph Laplacian L, allowing us to compute U_m. Finally, we can describe the computation that the layers perform, one after the other:

Start with the original features.

$$h^{(0)} = x$$

□ Learnable parameters.

Then iterate, for $k = 1, 2, \ldots$ upto K:

$$\hat{h}^{(k-1)} = U_m^T h^{(k-1)}$$

Convert previous feature $h^{(k-1)}$ to its spectral representation $\hat{h}^{(k-1)}$.

$$\hat{g}^{(k)} = \hat{w}^{(k)} \odot \hat{h}^{(k-1)}$$

Convolve with filter weights $\hat{w}^{(k)}$ in the spectral domain to get $\hat{g}^{(k)}$.
$\odot$ represents element-wise multiplication.

$$g^{(k)} = U_m \hat{g}^{(k)}$$

Convert $\hat{g}^{(k)}$ back to its natural representation $g^{(k)}$.

$$h^{(k)} = \sigma\left(g^{(k)}\right)$$

Apply a non-linearity σ to $g^{(k)}$ to get $h^{(k)}$.

The method above generalizes easily to the case where each $h^{(k)} \in \mathbb{R}^{d_k}$, as well: see for details.

With the insights from the previous section, we see that convolution in the spectral-domain of graphs can be thought of as the generalization of convolution in the frequency-domain of images.

Spectral Convolutions are Node-Order Equivariant

We can show spectral convolutions are node-order equivariant using a similar approach as for Laplacian polynomial filters.

Details for the Interested Reader

As in our proof before, let's fix an arbitrary node-order. Then, any other node-order can be represented by a permutation of this original node-order. We can associate this permutation with its permutation matrix P. Under this new node-order, the quantities below transform in the following way:

$$\begin{aligned} x &\to Px \\ A &\to PAP^T \\ L &\to PLP^T \\ U_m &\to PU_m \end{aligned}$$

which implies that, in the embedding computation:

$$\begin{aligned} &\hat{x} \to (PU_m)^T (Px) = U_m^T x = \hat{x} \\ &\hat{w} \to (PU_m)^T (Pw) = U_m^T w = \hat{w} \\ &\hat{g} \to \hat{g} \\ &g \to (PU_m)\hat{g} = P(U_m \hat{g}) = Pg \end{aligned}$$

Hence, as σ is applied elementwise:

$$f(Px) = \sigma(Pg) = P\sigma(g) = Pf(x)$$

as required. Further, we see that the spectral quantities $\hat{x}$, $\hat{w}$ and $\hat{g}$ are unchanged by permutations of the nodes. Formally, they are what we would call node-order invariant.

The theory of spectral convolutions is mathematically well-grounded; however, there are some key disadvantages that we must talk about:

- We need to compute the eigenvector matrix U_m from L. For large graphs, this becomes quite infeasible.
- Even if we can compute U_m, global convolutions themselves are inefficient to compute, because of the repeated multiplications with U_m and U_m^T.
- The learned filters are specific to the input graphs, as they are represented in terms of the spectral decomposition of input graph Laplacian L. This means they do not transfer well to new graphs which have significantly different structure (and hence, significantly different eigenvalues) .

While spectral convolutions have largely been superseded by 'local' convolutions for the reasons discussed above, there is still much merit to understanding the ideas behind them. Indeed, a recently proposed GNN model called Directional Graph Networks actually uses the Laplacian eigenvectors and their mathematical properties extensively.

Global Propagation via Graph Embeddings

A simpler way to incorporate graph-level information is to compute embeddings of the entire graph by pooling node (and possibly edge) embeddings, and then using the graph embedding to update node embeddings, following an iterative scheme similar to what we have looked at here. This is an approach used by Graph Networks . We will briefly discuss how graph-level embeddings can be constructed in Pooling. However, such approaches tend to ignore the underlying topology of the graph that spectral convolutions can capture.

Learning GNN Parameters

All of the embedding computations we've described here, whether spectral or spatial, are completely differentiable. This allows GNNs to be trained in an end-to-end fashion, just like a standard neural network, once a suitable loss function $\mathcal{L}$ is defined:

- **Node Classification**: By minimizing any of the standard losses for classification tasks, such as categorical cross-entropy when multiple classes are present:

$$\mathcal{L}(y_v, \hat{y_v}) = -\sum_c y_{vc} \log \hat{y_{vc}}.$$

  where $\hat{y_{vc}}$ is the predicted probability that node v is in class c. GNNs adapt well to the semi-supervised setting, which is when only some nodes in the graph are labelled. In this setting, one way to define a loss $\mathcal{L}_G$ over an input graph G is:

$$\mathcal{L}_G = \frac{\sum\limits_{v \in \text{Lab}(G)} \mathcal{L}(y_v, \hat{y_v})}{|\text{Lab}(G)|}$$

  where, we only compute losses over labelled nodes $\text{Lab}(G)$.

- **Graph Classification**: By aggregating node representations, one can construct a vector representation of the entire graph. This graph representation can be used for any graph-level task, even beyond classification. See Pooling for how representations of graphs can be constructed.

- **Link Prediction**: By sampling pairs of adjacent and non-adjacent nodes, and use these vector pairs as inputs to predict the presence/absence of an edge. For a concrete example, by minimizing the following 'logistic regression'-like loss:

$$\begin{aligned}\mathcal{L}(y_v, y_u, e_{vu}) &= -e_{vu} \log(p_{vu}) - (1 - e_{vu}) \log(1 - p_{vu}) \\ p_{vu} &= \sigma(y_v^T y_u)\end{aligned}$$

  where σ is the sigmoid function, and $e_{vu} = 1$ iff there is an edge between nodes v and u, being 0 otherwise.

- **Node Clustering**: By simply clustering the learned node representations.

The broad success of pre-training for natural language processing models such as ELMo and BERT has sparked interest in similar techniques for GNNs . The key idea in each of these papers is to train GNNs to predict local (eg. node degrees, clustering coefficient, masked node attributes) and/or global graph properties (eg. pairwise distances, masked global attributes).

Another self-supervised technique is to enforce that neighbouring nodes get similar embeddings, mimicking random-walk approaches such as node2vec and DeepWalk :

$$L_G = \sum_v \sum_{u \in N_R(v)} \log \frac{\exp z_v^T z_u}{\sum\limits_{u'} \exp z_{u'}^T z_u}.$$

where $N_R(v)$ is a multi-set of nodes visited when random walks are started from v. For large graphs, where computing the sum over all nodes may be computationally expensive, techniques such as Noise Contrastive Estimation are especially useful.

Conclusion and Further Reading

While we have looked at many techniques and ideas in this article, the field of Graph Neural Networks is extremely vast. We have been forced to restrict our discussion to a small subset of the entire literature, while still communicating the key ideas and design principles behind GNNs. We recommend the interested reader take a look at for a more comprehensive survey.

We end with pointers and references for additional concepts readers might be interested in:

GNNs in Practice

It turns out that accomodating the different structures of graphs is often hard to do efficiently, but we can still represent many GNN update equations using as sparse matrix-vector products (since generally, the adjacency matrix is sparse for most real-world graph datasets.) For example, the GCN variant discussed here can be represented as:

$$h^{(k)} = D^{-1}A \cdot h^{(k-1)} {W^{(k)}}^T + h^{(k-1)} {B^{(k)}}^T.$$

Restructuring the update equations in this way allows for efficient vectorized implementations of GNNs on accelerators such as GPUs.

Regularization techniques for standard neural networks, such as Dropout , can be applied in a straightforward manner to the parameters (for example, zero out entire rows of $W^{(k)}$ above). However, there are graph-specific techniques such as DropEdge that removes entire edges at random from the graph, that also boost the performance of many GNN models.

Different Kinds of Graphs

Here, we have focused on undirected graphs, to avoid going into too many unnecessary details. However, there are some simple variants of spatial convolutions for:

- Directed graphs: Aggregate across in-neighbourhood and/or out-neighbourhood features.
- Temporal graphs: Aggregate across previous and/or future node features.
- Heterogeneous graphs: Learn different aggregation functions for each node/edge type.

There do exist more sophisticated techniques that can take advantage of the different structures of these graphs: see for more discussion.

Pooling

This article discusses how GNNs compute useful representations of nodes. But what if we wanted to compute representations of graphs for graph-level tasks (for example, predicting the toxicity of a molecule)?

A simple solution is to just aggregate the final node embeddings and pass them through another neural network PREDICT_G:

$$h_G = \text{PREDICT}_G \Big(\text{AGG}_{v \in G} \left(\{ h_v \} \right) \Big)$$

However, there do exist more powerful techniques for ‘pooling’ together node representations:

- SortPool: Sort vertices of the graph to get a fixed-size node-order invariant representation of the graph, and then apply any standard neural network architecture.
- DiffPool: Learn to cluster vertices, build a coarser graph over clusters instead of nodes, then apply a GNN over the coarser graph. Repeat until only one cluster is left.
- SAGPool: Apply a GNN to learn node scores, then keep only the nodes with the top scores, throwing away the rest. Repeat until only one node is left.

Supplementary Material

Reproducing Experiments

The experiments from Spectral Representations of Natural Images can be reproduced using the following Colab notebook: Spectral Representations of Natural Images.

Recreating Visualizations

To aid in the creation of future interactive articles, we have created ObservableHQ notebooks for each of the interactive visualizations here:

- Neighbourhood Definitions for CNNs and GNNs
- Graph Polynomial Convolutions on a Grid
- Graph Polynomial Convolutions: Equations
- Modern Graph Neural Networks: Equations
- Modern Graph Neural Networks: Interactive Models which pulls together the following standalone notebooks:
  - Graph Convolutional Networks
  - Graph Attention Networks
  - GraphSAGE
  - Graph Isomorphism Networks
- Laplacian Eigenvectors for Grids
- Spectral Decomposition of Natural Images
- Spectral Convolutions: Equations

Acknowledgments

We are deeply grateful to ObservableHQ, a wonderful platform for developing interactive visualizations. The static visualizations would not have been possible without Inkscape and Alexander Lenail's Neural Network SVG Generator. The molecule diagrams depicted above were obtained and modified from Wikimedia Commons, available in the public domain.

We would like to acknowledge the following Distill articles for inspiration on article design:

- Visualizing memorization in RNNs
- Understanding RL Vision

We would like to thank Thomas Kipf for his valuable feedback on the technical content within this article.

We would like to thank David Nichols for creating Coloring for Colorblindness which helped us improve the accessibility of this article's color scheme.

We would also like to acknowledge CS224W: Machine Learning with Graphs as an excellent reference from which the authors benefitted significantly.

Ashish Tendulkar from Google Research India provided significant feedback on the content within this article, helping its readability. He also helped with identifying the topics this article should cover, and with brainstorming the experiments here.

A
P
fr
G
R
h
u
ir
w
a
h
a
re

Finally, we would like to thank Anirban Santara, Sujoy Paul and Ansh Khurana from Google Research India for their help with setting up and running experiments.

Author Contributions

Ameya Daigavane drafted most of the text, designed experiments and created the interactive visualizations in this article. **Balaraman Ravindran** and **Gaurav Aggarwal** extensively guided the overall direction of the article, deliberated over the design and scope of experiments, provided much feedback on the interactive visualizations, edited the text, and described improvements to make the article more

Discussion and Review

Review 1 - Chaitanya K. Joshi
Review 2 - Nick Moran
Review 3 - Anonymous

References

1. **A Gentle Introduction to Graph Neural Networks**
   Sanchez-Lengeling, B., Reif, E., Pearce, A. and Wiltschko, A., 2021. Distill. DOI: 10.23915/distill.00033
2. **Graph Kernels**
   [HTML]
   Vishwanathan, S., Schraudolph, N.N., Kondor, R. and Borgwardt,

Updates and Corrections

accessible to readers.

K.M., 2010. Journal of Machine Learning Research, Vol 11(40), pp. 1201-1242.

3. **Node2vec: Scalable Feature Learning for Networks**

   [link]
   Grover, A. and Leskovec, J., 2016. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 855–864. Association for Computing Machinery. DOI: 10.1145/2939672.2939754
4. **DeepWalk: Online Learning of Social Representations**

   [link]
   Perozzi, B., Al-Rfou,

R. and Skiena, S., 2014. Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 701–710. Association for Computing Machinery. DOI: 10.1145/2623330.2623732

5. **Convolutional Networks on Graphs for Learning Molecular Fingerprints**

   [PDF] Duvenaud, D.K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A. and Adams, R.P., 2015. Advances in Neural Information Processing Systems,

If you see mistakes or want to suggest changes, please create an issue on GitHub.

Vol 28, pp. 2224-2232. Curran Associates, Inc.

6. **Neural Message Passing for Quantum Chemistry**

   [HTML] Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O. and Dahl, G.E., 2017. Proceedings of the 34th International Conference on Machine Learning, Vol 70, pp. 1263-1272. PMLR.
7. **Learning Convolutional Neural Networks for Graphs** Niepert, M., Ahmed, M. and Kutzkov, K., 2016. Proceedings of the 33rd

International Conference on International Conference on Machine Learning - Volume 48, pp. 2014–2023. JMLR.org.

8. **A Tutorial on Spectral Clustering**
   [PDF]
   Luxburg, U.v., 2007. CoRR, Vol abs/0711.0189.
9. **Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering**
   [PDF]
   Defferrard, M., Bresson, X. and Vandergheynst, P., 2016. Advances in Neural Information Processing Systems, Vol 29, pp. 3844-3852. Curran

Reuse

Associates, Inc.

10. **Wavelets on Graphs via Spectral Graph Theory** [link]
    Hammond, D.K., Vandergheynst, P. and Gribonval, R., 2011. Applied and Computational Harmonic Analysis, Vol 30(2), pp. 129 - 150. DOI: https://doi.org/10.1016/j.acha.2010.04.005
11. **Chebyshev Polynomials** [link]
    Mason, J. and Handscomb, D., 2002. CRC Press.
12. **Semi-Supervised Classification with Graph Convolutional Networks** [link]
    Kipf, T.N. and Welling, M., 2017. 5th International

Conference on Learning Representations (ICLR) 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net.

13. **Graph Attention Networks**

    [link]
    Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P. and Bengio, Y., 2018. International Conference on Learning Representations.
14. **Inductive Representation Learning on Large Graphs**

    [PDF]
    Hamilton, W., Ying, Z. and Leskovec, J., 2017. Advances in Neural Information Processing

Systems, Vol 30, pp. 1024-1034. Curran Associates, Inc.

15. **How Powerful are Graph Neural Networks?**

    [link]
    Xu, K., Hu, W., Leskovec, J. and Jegelka, S., 2019. International Conference on Learning Representations.

16. **Relational inductive biases, deep learning, and graph networks**

    [PDF]
    Battaglia, P.W., Hamrick, J.B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V.F., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A.,

Faulkner, R., Gulcehre, C., Song, H.F., Ballard, A.J., Gilmer, J., Dahl, G.E., Vaswani, A., Allen, K.R., Nash, C., Langston, V., Dyer, C., Heess, N., Wierstra, D., Kohli, P., Botvinick, M., Vinyals, O., Li, Y. and Pascanu, R., 2018. CoRR, Vol abs/1806.01261. **Citation**

17. **Spectral Networks and Locally Connected Networks on Graphs**

    [PDF]
    Bruna, J., Zaremba, W., Szlam, A. and LeCun, Y., 2014.

International Conference on Learning Representations (ICLR 2014), CBLS, April 2014.

18. **ImageNet: A Large-Scale Hierarchical Image Database** Deng, J., Dong, W., Socher, R., Li, L., Li, K. and Fei-Fei, L., 2009. CVPR09.
19. **On the Transferability of Spectral Graph Filters** Levie, R., Isufi, E. and Kutyniok, G., 2019. 2019 13th International conference on Sampling Theory and Applications (SampTA), Vol (), pp.

1-5. DOI: 10.1109/SampTA45681.2019.9030932

20. **Directional Graph Networks** Beaini, D., Passaro, S., Létourneau, V., Hamilton, W.L., Corso, G. and Liò, P., 2021.
21. **Deep contextualized word representations** Peters, M.E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K. and Zettlemoyer, L., 2018. Proc. of NAACL.
22. **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**

    [link] Devlin, J., Chang, M., Lee, K.

For attribution in academic contexts, please cite this work as

and Toutanova, K., 2019. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp. 4171-4186. Association for Computational Linguistics. DOI: 10.18653/v1/N19-1423

23. **Strategies for Pre-training Graph Neural Networks** [link]
Hu*, W., Liu*, B., Gomes, J., Zitnik, M., Liang, P., Pande, V. and

Leskovec, J., 2020. International Conference on Learning Representations.

24. **Multi-Stage Self-Supervised Learning for Graph Convolutional Networks on Graphs with Few Labeled Nodes**

 [link]
 Sun, K., Lin, Z. and Zhu, Z., 2020. The Thirty-Fourth AAAI Conferenc on Artificial Intelligenc AAAI 2020, The Thirty-Second Innovative Applicatior of Artificial Intelligenc Conferenc IAAI 2020, The Tenth AAAI Symposiur on Educationa Advances

```
Daigavane, et al., "Understanding Convolutions on Graphs", Distill, 2021.
```

in Artificial Intelligenc EAAI 2020, New York, NY, USA, February 7-12, 2020, pp. 5892-5899. AAAI Press.

25. **When Does Self-Supervisio Help Graph Convolutic Networks?**

    [PDF] You, Y., Chen, T., Wang, Z. and Shen, Y., 2020.

26. **Self-supervised Learning on Graphs: Deep Insights and New Direction**

    [PDF] Jin, W., Derr, T., Liu, H., Wang, Y., Wang, S., Liu,

Z. and Tang, J., 2020.

27. **Noise-Contrastiv Estimation of Unnormali Statistical Models, with Applicatio to Natural Image Statistics**

    [HTML] Gutmann, M.U. and Hyvärinen, A., 2012. Journal of Machine Learning Research, Vol 13(11), pp. 307-361.

28. **Learning word embeddings efficiently with noise-contrastive estimation**

    [PDF] Mnih, A. and Kavukcuoglu, K., 2013. Advances in Neural Information Processing Systems, Vol 26, pp.

2265-2273. Curran Associates, Inc. BibTeX citation

29. **A Comprehensive Survey on Graph Neural Networks**

    [link] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C. and Yu, P.S., 2020. IEEE Transactions on Neural Networks and Learning Systems, pp. 1-21. DOI: 10.1109/TNNLS.2020.2978386
30. **Graph Neural Networks: A Review of Methods and Applications**

    [PDF] Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu,

Z. and Sun, M., 2018. CoRR, Vol abs/1812.08434.

31. **Dropout: A Simple Way to Prevent Neural Networks from Overfitting**

    [HTML]
    Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R., 2014. Journal of Machine Learning Research, Vol 15(56), pp. 1929-1958.
32. **DropEdge: Towards Deep Graph Convolutional Networks on Node Classification**

    [link]
    Rong, Y., Huang, W., Xu, T. and Huang,

```
@article{daigavane2021understanding,
  auth
```

J., 2020. Internation Conferenc on Learning Representa

33. **An End-to-End Deep Learning Architectu for Graph Classificat**

    [link]
    Zhang, M., Cui, Z., Neumann, M. and Chen, Y., 2018. Proceeding of the Thirty-Second AAAI Conferenc on Artificial Intelligenc (AAAI-18), the 30th innovative Applicatio of Artificial Intelligenc (IAAI-18), and the 8th AAAI Symposiu on Educationa Advances in Artificial Intelligenc

```
or = {Daigavane, Ameya and Ravindran, Balaraman and Aggarwal, Gaurav},
title = {Understanding Convolutions on Graphs},
journal = {Distill},
year = {2021},
note = {https://distill.pub/2021/understanding-gnns},
doi = {10.23915/distill.00032}
}
```

(EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, pp. 4438-4445. AAAI Press.

34. **Hierarchic Graph Represent Learning with Differentia Pooling**

    [PDF] Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W. and Leskovec, J., 2018. Advances in Neural Informatio Processing Systems, Vol 31, pp. 4800-4810. Curran Associates Inc.
35. **Self-Attention Graph Pooling**

    [HTML] Lee, J., Lee, I.

and Kang, J., 2019. Proceeding of the 36th Internation Conferenc on Machine Learning, Vol 97, pp. 3734-3743. PMLR.

ISSN 2476-0757