# Stat 236 Final Project: Recent Advances in Graph Prediction

Franklyn H. Wang
franklyn_wang@college.harvard.edu

Romil Sirohi
rsirohi@college.harvard.edu

Annie Zhu
szhu@college.harvard.edu

Fall 2020

## 1 Introduction

Prediction tasks on network-structured data have many modern applications ranging from friend recommendations on social networks to modelling the contacts between protein molecules in protein-protein Interaction (PPI) networks. In recent years, the encouraging performance of neural networks to problems such as Natural Language Processing (NLP) with autoencoders and image classification tasks with Convolutional Neural Networks (CNNs) has led to the development of Graph Neural Networks (GNNs), which has quickly risen to be the state-of-the-art benchmarks for various results. In this project, we seek to do a literature review of the big ideas of recent developments in GNNs.

The main idea behind most GNNs is to recursively find a latent feature representation (a.k.a. an embedding in the latent feature space) of a node by aggregating over the representation of its neighbors in the previous iteration, which is then combined with its own old representation to generate its new representation.

One way of thinking of these architectures is that they represent a natural combination of the ideas in RNNs and CNNs. RNNs apply the same transformation repeatedly when iterating over a sequence, whereas CNNs capture the useful inductive bias of spatial locality. Graph neural networks are built from these two ingredients, as well as an extra inductive bias in that the aggregation functions are required to be permutation invariant in their inputs, which is not required in convolutional neural networks.

Graph prediction tasks can be roughly divided into three categories: node classification, link prediction, and graph classification, following the categorization made by [HFZ+20]. The approaches for these classes have heavy overlaps, and we have introduced many of the classic methods in class. For example, we discussed using graph-structure heuristics to do link predictions. We also considered block matrix methods for community detection, which can bu used for node classification and link prediction. The GNN approach involves first generating the latent node representations and then using these features as the input of another neural network for the appropriate task. For node classification, GNNs are end-to-end because these embeddings can be used directly. For link prediction and graph classification, we will define a function–usually one of sum, mean or concatenate-on the latent features of the relevant nodes, and then use another neural network or function to give the final output. In the graph classification case, this middle step of processing all node representation is usually known as the READOUT function for the graph embedding. Much of recent development in this field has been devoted to finding more expressive architectures for powerful embeddings that encodes as much information as possible, and we will explain some examples in this paper.

In more recent work, other alternatives have been proposed for particular graph prediction tasks that achieve even better performance than vanilla GNNs. [ZC18] proposes the SEAL model, which encodes the $h$-hop neighborhood of the target pair of nodes as a link representation. [HHS+20] further augmented GNNs by proposing the Correct and Smooth (C&S) framework for node classification, and it is hoped that this will be extended to other tasks in the future. We will discuss these two methods in the project and conclude with the future directions in GNNs as we see them.

## 2 Notation

Let $G = (V, E)$ denote the *undirected* graph (network) of interest, such that $|V| = n$ and $|E| = m$. We use $\boldsymbol{A}$ for the adjacency matrix of a graph, and $\boldsymbol{D}$ for the diagonal degree matrix. We define the normalized adjacency matrix to be $\boldsymbol{A}_N$ to be equal to $\boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$. We define the Laplacian to be $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$, and the normalized Laplacian $\boldsymbol{L}_N$ to be $\boldsymbol{I} - \boldsymbol{A}_N$. Let $d(u, v)$ denote the shortest path distance between $u, v \in V$. Let $\mathcal{N}(v)$ be the neighbors of $v \in V$, and let $G_S^h$ be the $h$-hop enclosing subgraph for $S \subset V$ induced from $G$ by the nodes $\{w \in V : \exists v \in S,\, d(w, v) \leq h\}$. $G_v^h$ is the $h$-hop neighborhood of $v \in V$. We use the $n \times f$ node information matrix $\boldsymbol{X}$ with row vectors $\boldsymbol{x}_v$ to record any explicit attributes not encoded in the graph.

## 3 Convolutional GNNs and variants

For each node $v$ in a graph, a GNN learns a latent representation $\boldsymbol{h}_v$. In a GNN, each $\boldsymbol{h}_v$ is initialized as $\boldsymbol{x}_v$ (which is possibly empty) and then updated according to the values of its neighbors at each iteration. For example, at the $k$-th iteration, we update it as

$$\boldsymbol{a}_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\left\{\boldsymbol{h}_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right), \quad \boldsymbol{h}_v^{(k)} = \text{COMBINE}^{(k)}\left(\boldsymbol{h}_v^{(k-1)}, \boldsymbol{a}_v^{(k)}\right).$$

Therefore, a $k$-layer convolutional GNN incorporates the information of the $k$-hop neighborhood of the node. In Section 5 we will show that the localized nature of GNNs is not as limiting as it seems.

The choices of AGGREGATE and COMBINE define the architecture of the GNN. For example, the pooling variant of GraphSAGE [HYL17] uses a max-pooling operation over all neighbors, i.e.

$$\begin{aligned}\boldsymbol{a}_v^{(k)} &= \max\left(\left\{\sigma\left(\boldsymbol{W}_{\text{pool}}\,\boldsymbol{h}_{u_i}^{(k-1)} + \boldsymbol{b}\right), \forall u_i \in \mathcal{N}(v)\right\}\right), \\ \boldsymbol{h}_v^{(k)} &= \sigma\left(\boldsymbol{W}^{(k)} \cdot \text{CONCAT}\left(\boldsymbol{h}_v^{(k)}, \boldsymbol{a}_v^{(k)}\right)\right)\end{aligned}$$

where $\sigma$ is the sigmoid function.

In GCN [KW17], the COMBINE and AGGREGATE steps are combined into

$$\boldsymbol{h}_v^{(k)} = \text{ReLU}\left(\text{MEAN}\{h_u^{(k-1)} | u \in \mathcal{N}(v) \cup \{v\}\} \cdot \boldsymbol{W}^{(k)}\right). \tag{1}$$

In general, there are two classes of convolutional GNNs, spectral and spatial, of which GCN and GraphSAGE are representative examples, respectively. We now delve deeper into the theory of each of these two categories. We use [WPC+19] as a guide for papers to look at and expand on the relevant details.

## 3.1 Spectral-based GNNs

Spectral-based GNNs derive the name from using the spectrum Laplacian of $G$ to incorporate the graph structure into the network. By projecting the explicit features of the vertices onto the basis consisting of the eigenvectors of the Laplacian, we are allowed to consider the features in the Euclidean space with $\boldsymbol{U}$ as the basis. We can then perform convolutions in this transformed space, which as we will see below induces sparse convolutional filters. More recent development has focused on simplifying the calculations using approximations.

### 3.1.1 Theory

We consider the normalized graph Laplacian matrix $\boldsymbol{L}_N$.

**Lemma 3.1.** *The Laplacian (and the normalized Laplacian) is symmetric positive semidefinite.*

*Proof.* One can show that $\boldsymbol{x}^\top \boldsymbol{L}\boldsymbol{x} = \sum_{(u,v)\in E}(x_u - x_v)^2$. $\square$

Let the eigendecomposition of $\boldsymbol{L}_N$ be

$$\boldsymbol{L}_N = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top, \boldsymbol{U}^\top\boldsymbol{U} = \boldsymbol{U}\boldsymbol{U}^\top = \boldsymbol{I}.$$

Consider a graph feature vector $\boldsymbol{x} \in \mathbb{R}^n$ with one value corresponding to each node. Define the *graph Fourier transform* (GFT) as $F(\boldsymbol{x}) = \boldsymbol{U}^\top\boldsymbol{x}$ with inverse $F^{-1}(\tilde{\boldsymbol{x}}) = \boldsymbol{U}\tilde{\boldsymbol{x}}$.

GFT provides a natural way to define a convolution between $\boldsymbol{x}$ and a *filter* $\boldsymbol{g} \in \mathbb{R}^n$ as

$$\boldsymbol{x} *_G \boldsymbol{g} = F^{-1}(F(\boldsymbol{x}) \odot F(\boldsymbol{g})) = \boldsymbol{U}(\boldsymbol{U}^\top\boldsymbol{x} \odot \boldsymbol{U}^\top\boldsymbol{g}),$$

where we denote by $\odot$ the element-wise product. Observe that if we reparametrize the filter with the $n \times n$ diagonal matrix $\boldsymbol{G}_\theta = \operatorname{diag}(\boldsymbol{U}^\top\boldsymbol{g})$, then we can simplify the above definition as the matrix product

$$\boldsymbol{x} *_G \boldsymbol{g} = \boldsymbol{U}(\boldsymbol{U}^\top\boldsymbol{x} \odot \boldsymbol{U}^\top\boldsymbol{g}) = \boldsymbol{U}(\boldsymbol{U}^\top\boldsymbol{g} \odot \boldsymbol{U}^\top\boldsymbol{x}) = \boldsymbol{U}(\operatorname{diag}(\boldsymbol{U}^\top\boldsymbol{g})\boldsymbol{U}^\top\boldsymbol{x}) = \boldsymbol{U}\boldsymbol{G}_\theta\boldsymbol{U}^\top\boldsymbol{x}.$$

### 3.1.2 The Spectral GNN

Spectral-based GNNs are all constructed based on the above convolution. In particular, the Spectral GNN framework in [BZSL14] considers the following $K$-layer network. We initialize $\boldsymbol{H}^{(0)} = \boldsymbol{X}$ with $f_0 = f$. The $k$-th layer inputs the $n \times f_{k-1}$ feature matrix $\boldsymbol{H}^{(k-1)} = \begin{bmatrix} \boldsymbol{h}^{(k-1)}_{:,1} & \cdots & \boldsymbol{h}^{(k-1)}_{:,f_{k-1}} \end{bmatrix}$ and outputs the $n \times f_k$ feature matrix $\boldsymbol{H}^{(k-1)}$ with column vectors

$$\boldsymbol{h}^{(k)}_{:,j} = \tau\left(\sum_{i=1}^{f_{k-1}} \boldsymbol{U}\boldsymbol{\Theta}^{(k)}_{i,j}\boldsymbol{U}^\top\boldsymbol{h}^{(k-1)}_{:,i}\right), 1 \le j \le f_k,$$

where the $n \times n$ diagonal matrix $\boldsymbol{\Theta}^{(k)}_{i,j}$ is a $\boldsymbol{G}_\theta$ style filter that denotes the contribution of the output from the $i$-th filter in the $(k-1)$-th layer to the $j$-th filter in the $k$-th layer, and $\tau$ is some standard nonlinear activation function. This neural network can be trained by backpropagation.

Unfortunately, there are many inconveniences with the Spectral GNN framework. The eigenbases is highly sensitive to the particular graph in question, so the calculations could be unstable. Meanwhile, the fitted filters $\boldsymbol{\Theta}$ are valid only for the given graph, so unlike CNN filters, we cannot easily transfer them to a different graph. Lastly, computing $U$ requires an eigendecomposition, which takes $O(n^3)$ time and is computationally expensive. Convolution with the filter $\boldsymbol{U}\boldsymbol{\Theta}\boldsymbol{U}^\top\boldsymbol{h}$ takes $O(n^2)$ due to the size of the matrices and vectors, which is also not ideal.

### 3.1.3 Approximations and variants

Since then, a number of alternative spectral-based GNNs have been proposed to shorten the run-time. One method, the ChebNet [TLY19], is to approximate the filter so that we can avoid computing $\boldsymbol{U}$ explicitly. In Fourier analysis, Chebyshev polynomials are often used to approximate kernels due to their relationship with sine and cosine functions, so we apply them here as well.

Recall that Chebyshev polynomials are defined as

$$\begin{aligned} &T_0(x) = 1, \quad T_1(x) = x, \\ &T_i(x) = 2xT_{i-1}(x) - T_{i-2}(x). \end{aligned} \tag{2}$$

These polynomials form an orthogonal basis. Therefore, since $\boldsymbol{G}_\theta$ is a diagonal matrix, we can approximate to the $p$-th order

$$\boldsymbol{G}_\theta \approx \sum_{i=0}^{p} \theta_i T_i(\tilde{\boldsymbol{\Lambda}}),$$

where $\tilde{\boldsymbol{\Lambda}} = 2\boldsymbol{\Lambda}/\lambda_{\max} - \boldsymbol{I}_n$, $\tilde{\boldsymbol{L}}_N = 2\boldsymbol{L}_N/\lambda_{\max} - \boldsymbol{I}_n$ is a rescaling of the eigenvalues of $\boldsymbol{L}_N$ such that they are all in $[-1, 1]$. We now aim to learn the parameters $\boldsymbol{\theta} = \begin{bmatrix}\theta_0 & \dots & \theta_p\end{bmatrix}^\top$.

**Lemma 3.2.** *We claim that*

$$T_i(\tilde{\boldsymbol{L}}_N) = \boldsymbol{U} T_i(\tilde{\boldsymbol{\Lambda}}) \boldsymbol{U}^\top.$$

This can be proved with induction and the recursive formula for $T_i$. Therefore,

$$\boldsymbol{U}\boldsymbol{G}_\theta\boldsymbol{U}^\top\boldsymbol{h} \approx \sum_{i=0}^{p} \theta_i \boldsymbol{U} T_i(\tilde{\boldsymbol{\Lambda}})\boldsymbol{U}^\top \boldsymbol{h} = \sum_{i=0}^{p} \theta_i T_i(\tilde{\boldsymbol{L}}_N)\boldsymbol{h}.$$

We may then let $\tilde{\boldsymbol{h}}_i = T_i(\tilde{\boldsymbol{L}}_N)\boldsymbol{h}$, which we can also recursively compute using Eq. (2). This reduces the computation cost of $\boldsymbol{U}\boldsymbol{G}_\theta\boldsymbol{U}^\top\boldsymbol{h}$ to $O(m)$ because $\boldsymbol{L}_N$ is sparse with $O(m)$ nonzero entries and we can treat $p$ as a constant.

In addition to shorter run time, an additional advantage of the ChebNet is that it is localized to the $p$-hop neighborhood. Observe that $h_{v,j}$ depends on $h_{u,j}$ through the term

$$(\boldsymbol{U}\boldsymbol{G}_\theta\boldsymbol{U}^\top)_{v,u} \approx \left(\sum_{i=0}^{p} \theta_i T_i(\tilde{\boldsymbol{L}}_N)\right)_{v,u} \overset{\text{reparam.}}{=} \left(\sum_{i=0}^{p} c_i \tilde{\boldsymbol{L}}_N^i\right)_{v,u}.$$

**Lemma 3.3.** *When $d(u, v) > k$, we have $\left(L^k\right)_{v,u} = 0$.*

By the above lemma, if $u$ is not in the $p$-hop neighborhood of $u$, then the network does not factor in the influence of $p$ on $u$, so we can extract features about the local structure of the graph without worrying about the total size of the graph.

The GCN [KW17] is a further simplification of the ChebNet by restricting our approximation of the convolution to the first two Chebyshev polynomials, $T_0, T_1$. Then our sum becomes

$$\boldsymbol{U}\boldsymbol{G}_\theta\boldsymbol{U}^\top\boldsymbol{h} \approx [\theta_0 T_0(2\boldsymbol{L}_N/\lambda_{max} - \boldsymbol{I}_n) + \theta_1 T_1(2\boldsymbol{L}_N/\lambda_{max} - \boldsymbol{I}_n)]\boldsymbol{h}.$$

We take $\lambda_{max} = 2$, and further constrain that $\theta = \theta_0 = -\theta_1$. Then we have

$$\boldsymbol{U}\boldsymbol{G}_\theta\boldsymbol{U}^\top\boldsymbol{h} \approx [\theta - \theta(2\boldsymbol{L}_N/\lambda_{max} - \boldsymbol{I}_n)]\boldsymbol{h} = \theta(\boldsymbol{I} + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2})\boldsymbol{h}.$$

after some simplification.

Repeated multiplication with $\boldsymbol{I} + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$ can induce numerical instability. Empirically, we define instead

$$\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}, \tilde{\boldsymbol{D}} = \operatorname{diag}(\sum_j \tilde{A}_{ij}),$$

so that $\boldsymbol{I} + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2} \approx \tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}$.

We can generalize this into a convolutional layer, defined as

$$\boldsymbol{h}_{:,j}^{(k)} = \tau\left(\sum_{i=1}^{f_{k-1}} \theta_{ij}^{(k)} \tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}\boldsymbol{h}_{:,i}^{(k-1)}\right),$$
$$\boldsymbol{H}^{(k)} = \tau\left(\tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}\boldsymbol{H}^{(k-1)}\boldsymbol{\Theta}^{(k)}\right).$$

Observe that for each row $v \in V$, $\left(\tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}\right)_{v,:}\boldsymbol{H}^{(k-1)}$ averages over the rows $v \cup \mathcal{N}(v)$ of $\boldsymbol{H}^{(k-1)}$, allowing us to obtain the simplified form in Eq. (1). To this end, GCN can also be viewed as a spatial GNN, as we will see below.

Other types of spectral-based GNNs abound. For instance, rather than using Chebyshev polynomials, we could instead use Cayley polynomials, and we would end up with another type of graph neural network.

## 3.2 Spatial GNNs

Rather than incorportating the graph through the eigenbasis of the Laplacian, we can take advantage of its structure directly. Similar to CNNs on images, which convolve adjacent pixels, we can similarly convolve a node with its neighbors. See Fig. 1.

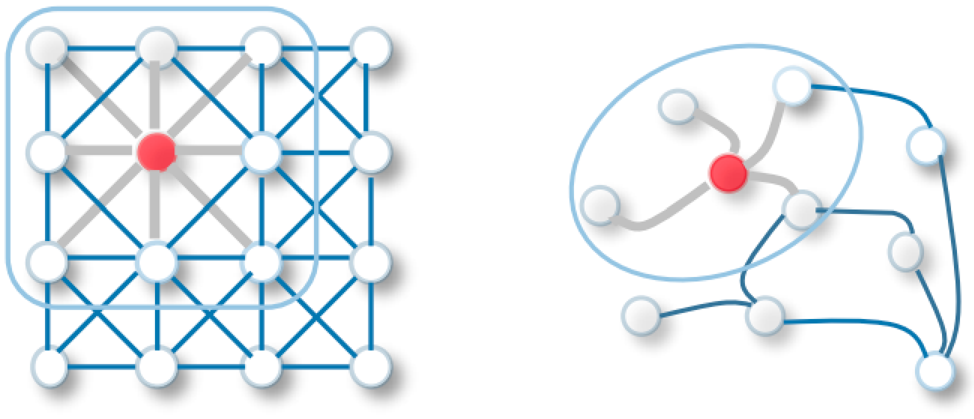

Figure 1: CNN vs Spatial GNN [May]

One of the first examples of such a framework is called neural networks for graphs (NN4G) [Mic09]. In matrix form, we again initialize $\boldsymbol{H}^{(0)} = \boldsymbol{X}$. After that, each layer is given by

$$\boldsymbol{H}^{(k)} = \sigma\left(\boldsymbol{X}\boldsymbol{W}^{(k)} + \sum_{i=1}^{k-1} \boldsymbol{A}\boldsymbol{H}^{(k-1)}\boldsymbol{\Theta}^{(i,k)}\right).$$

This looks quite similar to what we've seen above, but note that we are no longer in the Fourier space. We could simplify the neural network by setting $\boldsymbol{\Theta}^{(i,k)} = \boldsymbol{0}$ for all but $i = k - 1$, which would make it resemble more the neural networks that we typically see today.

Another spatial network approach is inspired by information transfer or diffusion along the edges. A diffusion convolutional neural network [AT16] assumes that each convolution is a step in a diffusion process. In this framework, each node diffuses to its neighbors with probability inversely proportional to its number of neighbors. The probability matrix is given by

$$\boldsymbol{P} = \boldsymbol{D}^{-1}\boldsymbol{A},$$

so that if the original input to the network is $X$, then its diffused version at the $k$-th timestep is $\boldsymbol{P}^k\boldsymbol{X}$, so that

$$\boldsymbol{H}^{(k)} = \tau(\boldsymbol{W}^{(k)} \odot \boldsymbol{P}^K\boldsymbol{X}),$$

and concatenates $\boldsymbol{H}^{(1)}, \ldots, \boldsymbol{H}^{(K)}$ into a tensor as the final outputs for the node/graph embeddings.

A diffusion graph convolution [LYSL17] is very similar. Rather than concatenating the output, it takes a sum. It also uses weights of a different dimension, defining each layer as

$$\boldsymbol{H}^{(k)} = \tau\left(\boldsymbol{P}^k\boldsymbol{X}\boldsymbol{W}^{(k)}\right),$$

so that the final output is $\boldsymbol{H} = \sum_{k=1}^K \boldsymbol{H}^{(k)}$. This is analogous to computing the stationary distribution of the diffusion process, which typically involves the power series of the probability transition matrix. Both of these approaches use $\boldsymbol{P}^k$, so the weight of nodes $k$ steps apart decays exponentially.

Alternatively, we could consider the length of the shortest path to boost the contribution of distant nodes [TNS18]. Consider a binary matrix $\boldsymbol{S}^{(k)}$ where $S^{(k)}_{u,v} = 1$ if $d(u,v) = k$, and 0 otherwise. At each layer, we convolve over pairs of nodes with the same distance at a time. Consider

$$\operatorname{diag}\left(\sum_{l=1}^{n} S^{(j)}_{i,l}\right)^{-1} \boldsymbol{S}^{(j)}\boldsymbol{H}.$$

The $v$-th row of the product is the average of rows $u$ of $\boldsymbol{H}$ such that $d(u,v) = j$. The GNN has

$$\boldsymbol{H}^{(k)}_j = \sigma\left(\sum_{t=0}^{r} \operatorname{diag}\left(\sum_{l=1}^{n} S^{(j)}_{i,l}\right)^{-1} \boldsymbol{S}^{(j)}\boldsymbol{H}^{(k-1)}_t\boldsymbol{W}^{(k)}_{t,j}\right).$$

There are similar approaches that avoid the costly computation of $\{S^{(j)}\}$, as the best known all shortest path algorithm takes $O(n^3)$. This approach is called a partition graph construction, which splits the graph split into neighborhoods, each with their own adjacency matrix. The convolution is then taken within each group at a time. We will not go into the details.

Another class of spatial convolutional neural networks is called a message passing neural net (MPNN) [GSR+17]. We can think of information as being passed along edges. The information about each neighbor is aggregated at each step:

$$\sum_{u\in\mathcal{N}(v)} M_k(\boldsymbol{h}^{(k-1)}_v, \boldsymbol{h}^{(k-1)}_u, \boldsymbol{e}_{vu})$$

for some function $M_k$, where $\boldsymbol{x}_{v,u}$ encodes the explicit attributes of the edge $(v,u)$ should they exit. This is then combined with the parameters of the current iteration to give the entire convolution as

$$\boldsymbol{h}^{(k)}_v = U_k\left(\boldsymbol{h}^{(k-1)}_v, \sum_{u\in N(v)} M_k(\boldsymbol{h}^{(k-1)}_v, \boldsymbol{h}^{(k-1)}_u, \boldsymbol{e}_{vu})\right).$$

The output is $R\left(\{h_v^{(K)}\}\right)$ for some function $R$. This framework can be applied to many of the neural nets we've already seen.

Unfortunately, these message passing neural nets often have some deficiencies in their ability to distinguish graphs. To fix this, [XHLJ19] one can add $\epsilon$ extra weighting on the center node. A graph isomorphism network, for example, has this structure

$$\boldsymbol{h}_v^{(k)} = MLP\left((1+\epsilon^{(k)})\boldsymbol{h}_v^{(k-1)} + \sum \boldsymbol{h}_u^{(k-1)}\right),$$

where the $MLP$ function is a learnable multi-layer perceptron.

## 4 Power of GNNs

One reason why GNNs have attracted interest is that many algorithms on graphs are natural fits for the GNN framework. Consider the following figure.

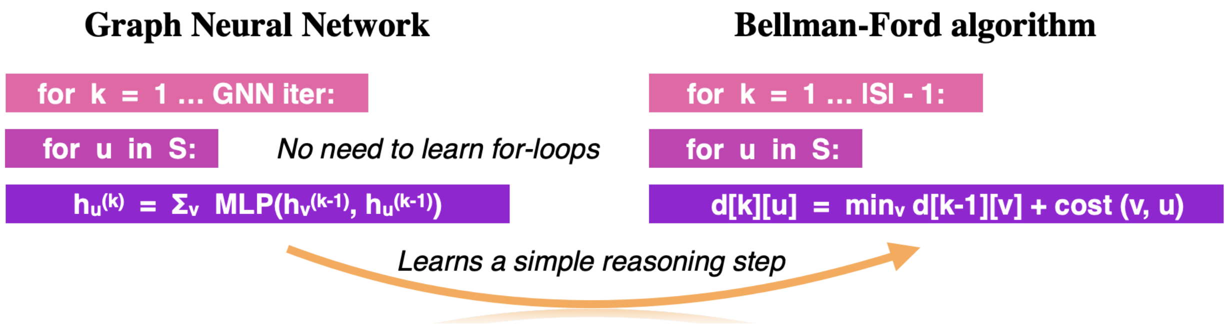
Graph Neural Network
Bellman-Ford algorithm
for k = 1 … GNN iter:
for u in S:
No need to learn for-loops
h_u^(k) = Σ_v MLP(h_v^(k-1), h_u^(k-1))
for k = 1 … |S| - 1:
for u in S:
d[k][u] = min_v d[k-1][v] + cost (v, u)
Learns a simple reasoning step

Figure 2: Bellman Ford is naturally captured by a GNN [XHLJ19]

This shows how the classical Bellman-Ford algorithm for finding shortest paths is captured by GNNs. One upshot of this is that graph neural networks may offer a novel way of addressing the classical NP-hard problems in graph theory – although it is interesting that the state of the art results in this field also require a post-processing step [KL20].

This discussion serves to indicate that the inductive biases of GNNs may be useful. However, inductive biases must come with restrictions, the lesson of the No Free Lunch Theorem. What is the extent of these restrictions? This is the question answered by [XHLJ19]. With normal fully connected networks, certain universal approximation theorems are known which imply a sufficiently large neural network can approximate any arbitrary function. This paper shows that a similar results Namely, the paper shows that graph neural networks (in the form specified earlier) will yield the same . that are equivalent under the Weisfeller-Lehmann Test. The Weifeller Lehmann test can be thought of as the ultimate aggregation of all neighborhood information - inspired by this, the second contribution of [XHLJ19] is to show that one can create a graph neural network which is *as strong* as the Weisfeller-Lehmann Test – which they call the Graph Isomorphism Network – and that this achieves state of the art accuracies on many datasets.

Another work which characterizes the expressive power of graph neural networks is [XLZ+19]. They find that the above observed phenomenon of *algorithmic alignment* does indeed serve as a useful predictor of GNN performance.

## 5 GNN for link prediction

As we have shown above, convolutional GNNs have been shown to have very good performance in using the the latent feature representations for node classification. We now move on to study the pros and cons of using GNNs for link prediction, i.e. predicting for $u, v \in V$ whether $(u, v) \in E$.

### 5.1 GNN approximates heuristics

Before the era of GNNs, a large class of methods for link prediction is to use heuristics, many of which we have studied in class. Heuristics aim to capture the similarity between $u$ and $v$. We define a $k$-th order heuristic as those that can be calculated from the $k$-hop neighborhoods $\mathcal{N}_u$ and $\mathcal{N}_v$. In general, low-order heuristics capture the "local similarity" of $u$ and $v$ because they only involve the immediate neighborhood, whereas high-order heuristics capture the "global similarity" of $u$ and $v$ via path counting. For example,

- First-order heuristics: common neighbors, preferential attachment.
- Second-order heuristics: Adamic-Adar.
- High-order heuristics: Katz index, PageRank, average commute time.

Judging from the structure of the GNN, one may suppose that the depth of GNNs impose a limit on the highest-order heuristic that it can accommodate. However, most well-known high-order heuristics discount the structure of the graph farther away from the target nodes $u$ and $v$. Formally, [ZLX+20] defines a *$\gamma$-decaying heuristic* such that for some $\gamma \in (0, 1)$ and $f(u, v, \ell) \geq 0$,

$$\mathcal{H}(u, v) = \eta \sum_{\ell=1}^{\infty} \gamma^{\ell} f(u, v, \ell).$$

**Theorem 5.1.** *If a $\gamma$-decaying heuristic $\mathcal{H}(u, v) = \eta \sum_{\ell=1}^{\infty} \gamma^{\ell} f(u, v, \ell)$. satisfies*

- *there exists $0 < \lambda < \gamma^{-1}$ such that $f(u, v, \ell) \leq \lambda^{\ell}$;*
- *there exists integer-valued function $g(h) = \Omega(h)$ such that for all $1 \leq h \leq g(h)$, $f(u, v, \ell)$ is calculable from the h-hop enclosing subgraph $G_{u,v}^{h}$.*

*then $\mathcal{H}(u, v)$ can be approximated from $G_{u,v}^{h}$ with an error that decreases exponentially with h.*

*Proof.* Since $\lambda\gamma < 1$, we can bound

$$\left| \mathcal{H}(u, v) - \eta \sum_{\ell=1}^{g(h)} \gamma^{\ell} f(u, v, \ell) \right| = \eta \sum_{\ell=g(h)+1}^{\infty} \gamma^{\ell} f(u, v, \ell) \leq \eta \sum_{\ell=g(h)+1}^{\infty} \gamma^{\ell} \lambda^{\ell} = \eta \frac{(\gamma\lambda)^{g(h)+1}}{1 - \gamma\lambda},$$

which decreases exponentially with $h$ because $g(h) = \Omega(h)$. □

We finish by showing that the Katz index, defined as

$$\mathcal{K}(u, v) = \sum_{\ell=1}^{\infty} \beta^{\ell} \left( \#\text{path}_{u,v}^{(\ell)} \right) = \sum_{\ell=1}^{\infty} \beta^{\ell} \left( \boldsymbol{A}^{\ell} \right)_{u,v}$$

is a $\gamma$-decaying heuristic under weak conditions. Indeed, it suffices to take $\eta = 1$ and $\gamma = \beta$. Now, $g(h) = 2h + 1$ satisfies the second property because every path of length $\leq 2h + 1$ must only contain points that are of distance at most $h$ to one of the endpoints. Meanwhile, we consider the following lemma

**Lemma 5.2.** *If $d$ is the maximum degree of the network $G$, then for all $u, v \in V$,*

$$\left(\boldsymbol{A}^{\ell}\right)_{u,v} \leq d^{\ell}.$$

*Proof.* We use induction. The base case $\ell = 1$ is obvious. For the inductive step, if $\left(\boldsymbol{A}^{\ell}\right)_{u,v} \leq d^{\ell}$. for all $u, v \in V$, then

$$\left(\boldsymbol{A}^{\ell+1}\right)_{u,v} = \sum_{k=1}^{n} \left(\boldsymbol{A}^{\ell}\right)_{u,k} \boldsymbol{A}_{k,v} \leq d^{\ell} \sum_{k=1}^{n} \boldsymbol{A}_{k,v} \leq d^{\ell} d = d^{\ell+1}.$$

□

Therefore as long as $d < \beta^{-1}$, the Katz-index is a $\gamma$-decaying sequence and can be well approximated by a GNN. Empirically, this is usually satisfied because we typically choose $\beta$ to be very small, i.e. on the order of $10^{-4}$.

One can show that the PageRank is also a $\gamma$-decaying sequence, but we omit the proof here.

The fact that GNN can approximate heuristics is very encouraging. On one hand, this implies that the superior performance of GNN to heuristics in link prediction tasks should not come as a surprise. On the other hand, the performance of the same heuristics can often vary a lot across data sets dependent due to the different underlying principles that generate these data sets. For instance, common neighbors could be a good heuristic for social networks, but it is less appropriate for PPI. Therefore, the flexibility of GNN allows us to leave it to the network to learn the correct mechanism for the particular data set, so that we can achieve the full potential of heuristics through approximation and outperform every fixed heuristic by not limiting ourselves to any particular one of them. The experiments in [ZC18] verify this claim.

## 5.2 Latent node representation vs link representation

Building from the GNNs that generate latent representations for the nodes, the common solution to link prediction is to combine the latent features of the two endpoints of the link in question with a simple function–usually concatenation–and fit an extra neural network to compute the link prediction. However, this method can be lacking for practical purposes. In particular, we would like to find a latent link representation that would allow us to tell apart $(v_1, v_2)$ vs $(v_1, v_3)$ in Fig. 3.

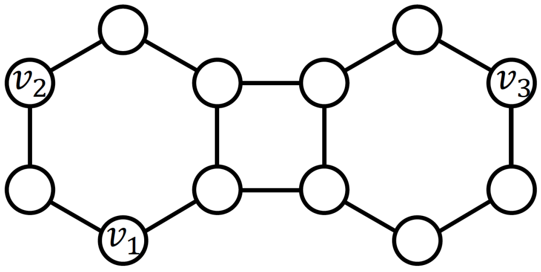

Figure 3: The latent node representation of $v_2$ and $v_3$ must be the same by symmetry, yet the intuitively, the predicted link probability for $(v_1, v_2)$ vs $(v_1, v_3)$ should be different [ZLX+20]

To combat this issue, [ZC18] proposes the SEAL framework. For each prediction on $(u, v)$, we modify the input to incorporate the structural information about the the $h$-hop enclosing subgraph $G^h_{(u,v)}$. More concretely, inspired by the Weifeller Lehman test, we construct a node labelling hashing function $L_{u,v} : V \to \mathbb{N}$ to encode the role of a node $w$ in the subgraph. Let $L_{u,v}(u) = L_{u,v}(v) = 1$, and

$$\forall w \in V - \{u, v\}, \quad L_{u,v}(w) = 1 + \min\left(d(u, w), d(v, w)\right) + \lfloor d/2 \rfloor [\lfloor d/2 \rfloor + (d\%2) - 1]$$

where we use the shorthand $d = d(u, w) + d(v, w)$ and let $(d\%2)$ represent the division remainder. Note that the function is symmetric with respect to $u$ and $v$. To gain some insight about this hashing function, here are its select values:

| $d(u,w)$ | $d(v,w)$ | $L_{u,v}(w)$ |
|---|---|---|
| 1 | 1 | 2 |
| 1 | 2 | 3 |
| 1 | 3 | 4 |
| 2 | 2 | 5 |
| 1 | 4 | 6 |
| 2 | 3 | 7 |

Table 1: Select mappings of the Double-Radius Node Labeling (DRNL) function [ZC18]

The guiding principle behind the labelling function is that we assign increasing labels to nodes in the order of the sum and product of its radius with respect to $u, v$, such that:

- if $d(w, u) + d(w, v) \neq d(x, u) + d(x, v)$, then $d(w, u) + d(w, v) < d(x, u) + d(x, v)$ if and only if $L_{u,v}(w) < L_{u,v}(x)$;
- if $d(w, u) + d(w, v) = d(x, u) + d(x, v)$, then $d(w, u)d(w, v) < d(x, u)d(x, v)$ if and only if $L_{u,v}(w) < L_{u,v}(x)$.

We incorporate this label into the GNN by concatenating each $L_{u,v}(w)$ to the vector of explicit attributes $\boldsymbol{x}_w$. We use the new node information matrix $\boldsymbol{X}_{u,v}$ as the input to our GNN and use one of the existing GNN architecture DGCNN. The experiments in [ZC18] show that SEAL outperforms most state-of-the-art latent feature methods thanks to the incorporation of the graph structure.

## 6 Correct & Smooth

Correct & Smooth is a general method that does not use any neural architectures. Instead, it's a fairly direct way of incorporating base predictions (obtained through any method) into finalized predictions, using the rule of thumb that generally models behave similarly on adjacent nodes. The idea is to use the training data twice. The first time is to estimate error that can be propagated along the edges; error should be correlated along neighboring vertices. The second time is to smooth out the final output, as the values of the training data are known for sure.

### 6.1 Generate base predictions

For the first step, base predictions are generated without using information about the edges, and just the training data. Some function $f$ is applied to $X$ and returns a base prediction $Z \in \mathbb{R}^{n \times c}$

which represent predictions for all nodes. Each row of $Z$ contains probabilities that a given node belongs to a given class.

## 6.2 Correct

In this step, we use the fact that model errors are often correlated between adjacent nodes. Thus, we propagate known error at training vertices along the edges. First we compute this error, defined on the training vertices $L_t$:

$$E_{L_t} = Z_{L_t} - Y_{L_t}$$

and $E$ is 0 elsewhere (on the validation vertices $L_v$ and unlabeled vertices $U$). We define

$$\hat{E} = \arg\min_{E'} \operatorname{tr}(E'^\top (I - N_A) E') + \mu |E' - E|_F^2 = \arg\min_{E'} \operatorname{tr}(E'^\top N_L E') + \mu |E' - E|_F^2 [1]$$

The first term encourages model errors on adjacent nodes to be similar to each other, the second term keeps $W$ close to our base estimate of the error, and the variable $\mu$ controls this trade off. It turns out this choice of propagation is ideal under some assumptions about normality.

After computing $\hat{E}$, we want the scale of the error to be correct. Auto-scale is a simple way to do that. We first compute $\sigma$, the average error magnitude for nodes in our training set. Then we add the scaled error back in

$$Z^{(r)}_{i,\cdot} = Z_{i,\cdot} + \sigma \hat{E}_{\cdot,i} / |\hat{E}^\top_{\cdot,i}|_1$$

## 6.3 Smooth

We now construct our best guess $G$. For training data (and validation data), this is the truth. Elsewhere, this is our guess from above.

$$G_{L_t} = Y_{L_t}, G_U = Z_U^{(r)}, G_{L_v} = Z_{L_v}$$

Our final prediction array is then given by

$$\hat{G} = \arg\min_{G'} \operatorname{Tr}(G'^\top (I - N_A) G') + \mu |G' - G|_F^2$$

Notice that this equation is quite similar to the equation used in "Correct", but it is solving for the final matrix, rather than an error matrix. We can use the same iterative methods for this optimization problem as the previous one.

## 6.4 Discussion

Interestingly, correct is known to only sometimes yield successful results, whereas smooth very often leads to strong results. The math above sheds light as to why that might happen. This can be seen from the assumptions – correct assumes that model errors are spatially correlated, where smooth only assumes that true labels are spatially correlated. For a perfect model, the first assumption would fail, and only the second one would remain true.

[1] This problem can be solved with an iterative method, by iterating

$$E^{(t+1)} = \frac{\mu}{1+\mu} E + \frac{1}{1+\mu} N_A E^{(t)}$$

where $E^{(0)} = E$.

## 7 Conclusion and Outlook

Graph Neural Networks have already achieved impressive results on many tasks relating to graph prediction. What comes next? From an optimistic perspective, papers like [HHS+20] and [KL20] suggest that neural networks are very good at unstrucured prediction tasks, they may have issues in problems that have more "hard constraints", but there we may be able to incorporate other approaches.

We are excited to see what comes from the next decade of graph prediction.

## References

[AT16] James Atwood and Don Towsley. Diffusion-convolutional neural networks. *Advances in neural information processing systems*, 29:1993–2001, 2016.

[BZSL14] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and deep locally connected networks on graphs. In *Proceedings of ICLR*, 2014.

[GSR+17] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

[HFZ+20] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[HHS+20] Qian Huang, Horace He, Abhay Singh, Ser-Nam Lim, and Austin R Benson. Combining label propagation and simple models out-performs graph neural networks. *arXiv preprint arXiv:2010.13993*, 2020.

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of NeurIPS*, 2017.

[KL20] Nikolaos Karalias and Andreas Loukas. Erdos goes neural: an unsupervised learning framework for combinatorial optimization on graphs. *Advances in Neural Information Processing Systems*, 33, 2020.

[KW17] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of ICLR*, 2017.

[LYSL17] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. *arXiv preprint arXiv:1707.01926*, 2017.

[May] Inneke Mayachita. Understanding graph convolutional networks for node classification.

[Mic09] Alessio Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

[TLY19] Shanshan Tang, Bo Li, and Haijun Yu. Chebnet: Efficient and stable constructions of deep neural networks with rectified power units using chebyshev approximations. *arXiv preprint arXiv:1911.05467*, 2019.

[TNS18] Dinh V Tran, Nicolò Navarin, and Alessandro Sperduti. On filter size in graph convolutional networks. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1534–1541. IEEE, 2018.

[WPC$^+$19] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *arXiv:1901.00596*, 2019.

[XHLJ19] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[XLZ$^+$19] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? *arXiv preprint arXiv:1905.13211*, 2019.

[ZC18] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Proceedings of NeurIPS*, 2018.

[ZLX$^+$20] Muhan Zhang Zhang, Pan Li, Yinglong Xia, Kai Wang, and Jin Long. Revisting graph neural networks for link prediction. *arXiv preprint arXiv:2010.16103*, 2020.