Recommender Systems with GNNs in PyG

Derrick Li · Follow

Published in Stanford CS224W: Machine Learning with Graphs · 19 min read · Jan 10, 2022

By Derrick Li, Peter Maldonado, Akram Sbaih as part of the Stanford CS224W (Machine Learning with Graphs) course project.

Introduction

It is finally winter break and you've got some free time, so you decide to binge watch some good films with your friends. They don't want to waste the break trying to make decisions on which movies, so they leave it to you to decide. With thousands of possible options and limited time, you struggle with choosing and wish you had someone to recommend you personally tailored movies. This is precisely the job of a recommender system, specifically the one we are building today.

Recommender systems have been a hot topic of research because they have very ubiquitous applications nowadays. The quantity of products and content available online is so overwhelming and rapidly growing that no one

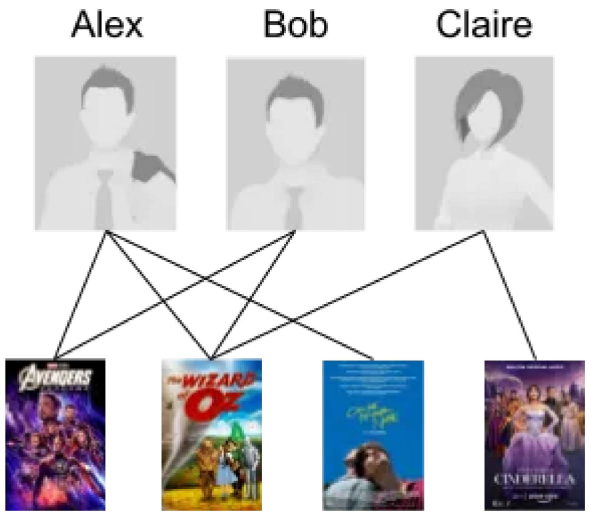
has the capacity to choose manually what's relevant for them. Recommender systems now show you top picks in music (Spotify), movies (Netflix), merchandise (Amazon), and social media posts (TikTok), all personalized to your interests without asking you any questions. So how do they figure you out?

Collaborative Filtering (CF) is the primary approach that recommender systems use to figure out your interests. It assumes that you would be interested in items that other similar users have shown interest in. In this context, similar users are other users who have historically interacted with mostly the same items that you interacted with.

For example, if Alex and Bob have both liked "Avengers" and "Wizard of Oz" and nothing else, and then Alex likes "Call Me by Your Name", CF will recommend "Call Me by Your Name" to Bob. Notice that there isn't necessarily a common genre for these three movies, but because Bob has historically liked the same things as Alex, it makes sense to recommend what Alex likes to Bob in the future, regardless of the content. Importantly, CF doesn't need to follow the algorithm I provided in this example. As we will see later in this article, CF can be the nebulous concept of predicting user interest in items based on similarity with historical user interest patterns.

A good way to visualize the interactions in a recommender system is using a **bipartite graph** with the users and items (movies in this case) as nodes, and edges between them indicating user-item interactions. Those interactions could be a user positively rating a recipe, or buying a product, or watching a video. The graph will be bipartite because users can be interested in items, but items and users can't be interested in other items or users, respectively. Look at the following graph visualization for the earlier example. Here, we're

trying to predict if Bob will like “Call Me by Your Name” or “Cinderella” more given the history of users previous likes (black edges). This graph representation enables us to employ powerful models, such as deep Graph Convolutional Networks (GCN) which we will discuss in this article.



User-Item interactions can be modeled as a bipartite undirected graph.

In this tutorial, we will explain how to build a movie recommender system by applying powerful Graph Neural Networks (GNNs), like NGCF and LightGCN, on these bipartite graph representations, and then compare them (spoiler alert: we will learn a general AI lesson along the way too: more complicated methods don’t necessarily perform better than simpler ones, but rather quite the opposite — stay tuned!).

Colab Notebook

Here is the link to the Google Colab notebook that implements the models in this post.

The Dataset

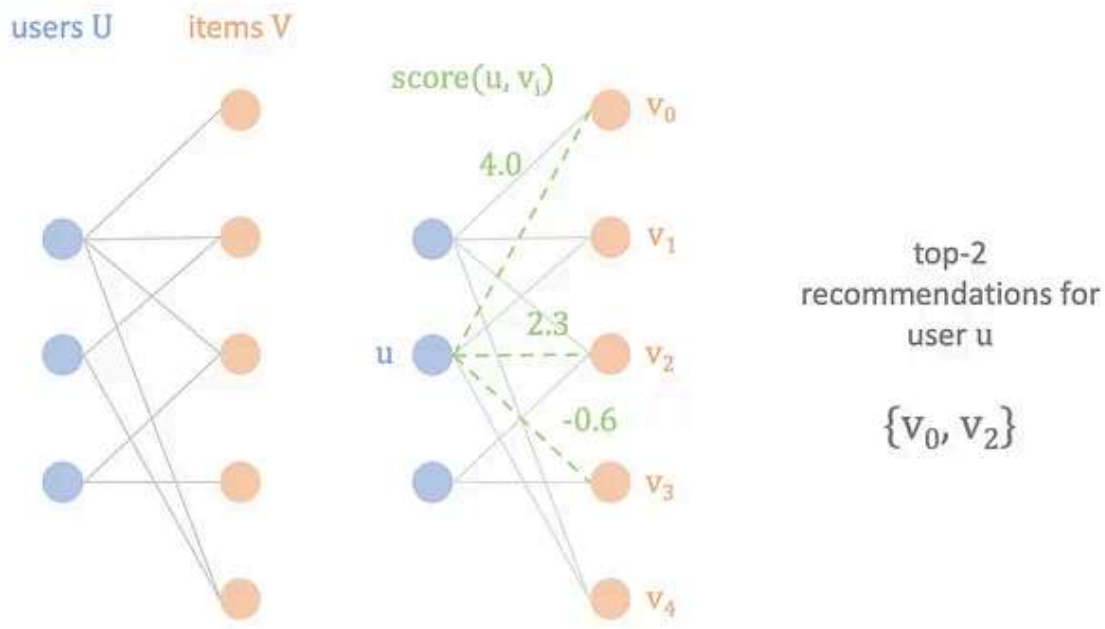
For this post, we'll be using the MovieLens 100K dataset, which contains 100,000 ratings by 943 users of 1682 items (movies). To ensure the quality of the dataset, each user has rated at least 20 movies (every user node has at least twenty edges).

We'll focus on the interactions between users and items, in this case user ratings of movies, but the dataset also provides metadata about users and movies, such as user demographics and movie titles, release dates, and genres. As alluded to above, the user ratings of movies create a bipartite graph, which we can apply graph machine learning methods to recommend new movies to users.

So what should a recommender system do?

If we're serious about building this movie recommendation service, then we need to be more formal about our objective.

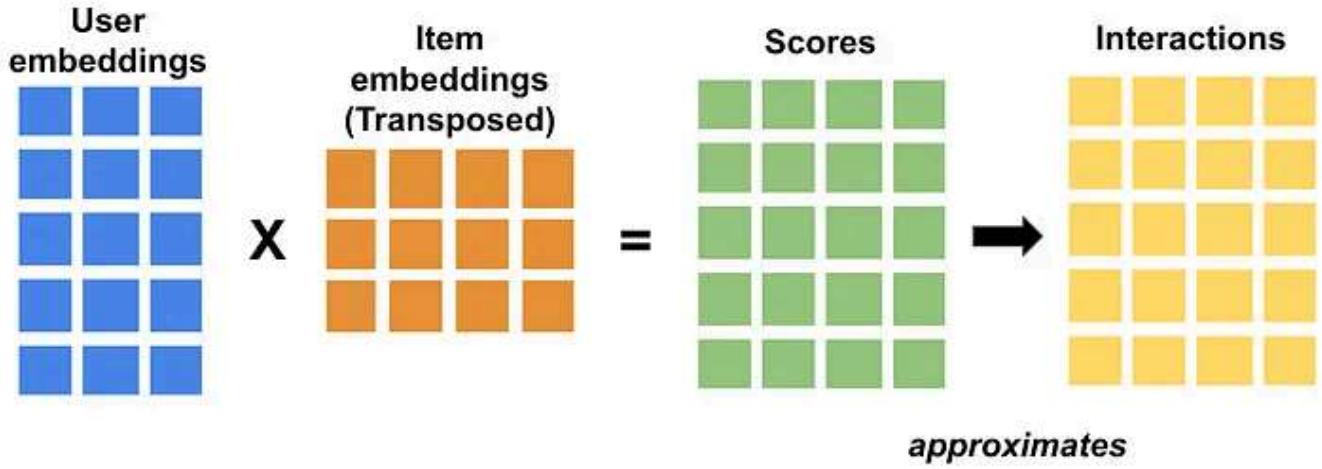
Given some set of users U and items I, we want to produce a real-valued scalar $score(u, i)$ for each user u and item i. For each user u we want to recommend the K items with the highest score that they have not yet interacted with. (K is a small positive-integer, typically between 10–100.)



This post is going to focus on embedding-based models–those that learn an embedding ***e_u*** for each user and ***e_v*** for each item. We then model ***score(u,v)*** = ***f_theta(u, v)*** where ***f*** is some function parameterized by parameters ***theta.*** The simplest embedding-based model sets ***f*** to be the inner product between the two feature vectors.

$$score(u, v) = f_\theta(u, v) = e_u^T e_v$$

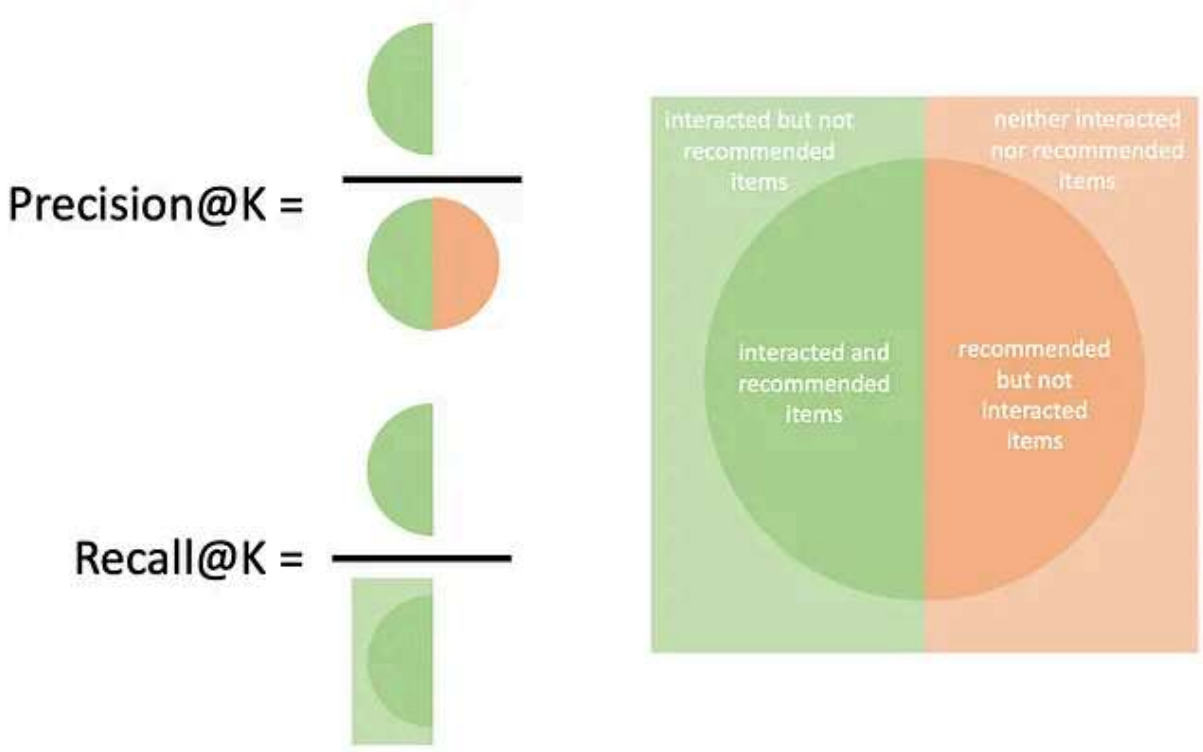
Let's visualize what this would look like for a matrix:



Then computing the scores for all users and items resolves to taking the product of the user embedding matrix and item embedding matrix. The GNN-based models we will discuss later in this post will still use this notion of user and item embeddings, but with a more complicated choice of the function f.

But how can we measure whether we are producing *good* recommendations? Two common metrics are **precision@K** and **recall@K**. To evaluate both metrics, we hold out a portion of seen user-item interactions.

- precision@K: the fraction of recommended items that are actually interacted with in the held-out user-item interactions set. In other words, how many recommended items are interacted with?
- recall@K: the fraction of held-out user-item interactions that are recommended. In other words, how many interacted items are recommended?



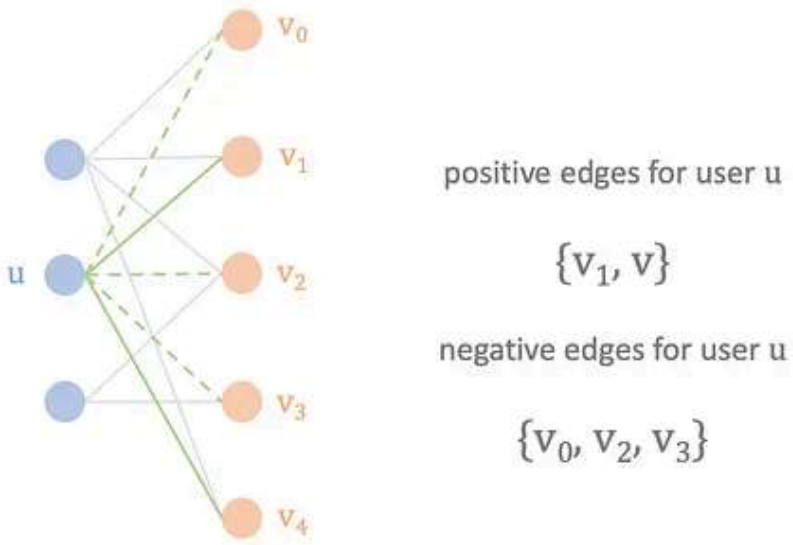
precision@K and recall@K use the same numerator but different denominators. The denominator of precision@K is the number of recommended items, K.

Note that in practice we can only optimize model parameters to achieve high precision@K and recall@K on *seen* user-item interactions, which we hope would lead to high performance on *unseen* interactions.

You might have noticed that, unfortunately, neither precision@K nor recall@K are differentiable. Instead, we use a surrogate loss function to enable efficient gradient-based optimization. Here, a surrogate loss function simply means a differentiable function that aligns well with the original training objective.

Specifically, we use the Bayesian Personalized Ranking (BPR) loss as the surrogate. To understand BPR, we need to define the notion of positive and negative edges: positive edges are those that exist in the graph and negative edges are those that don't. In our bipartite graph, we can define for user u

the set of all positive edges containing ***u***, ***E(u)*** and the set of all negative edges containing ***u***, ***E_neg(u)***.



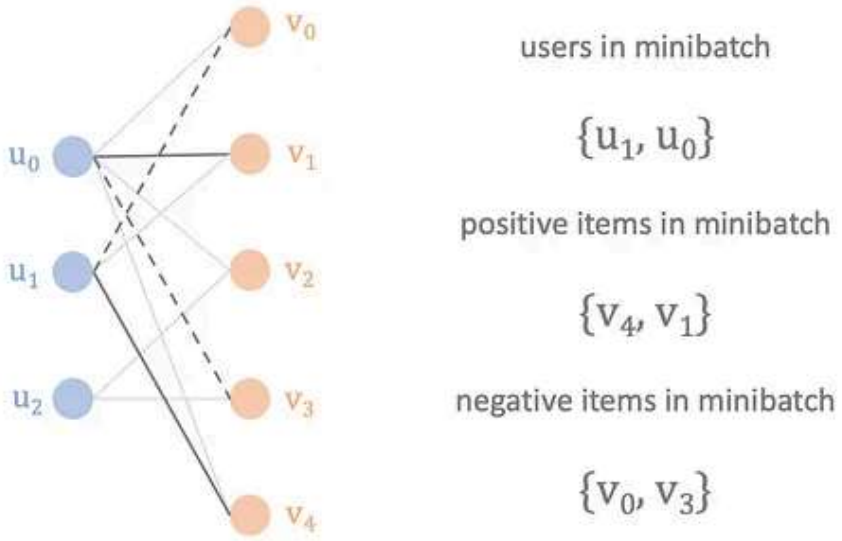
The BPR loss for a single user ***u*** is:

$$BPR(u) = \frac{1}{|E| \cdot |E_{neg}|} \sum_{(u,v_{pos}) \in E(u)} \sum_{(u,v_{neg}) \in E_{neg}(u)} -\log\left(\sigma(f_\theta(u, v_{pos}) - f_\theta(u, v_{neg}))\right)$$

And the final BPR loss is:

$$Loss = \frac{1}{|U|} \sum_{u \in U} BPR(u)$$

In order to efficiently estimate the BPR loss and optimize *f_theta*, we train the model by sampling minibatches. For each minibatch, we sample the set of users and for each user sample one positive item (the item from a positive edge containing the user in question) and one negative item.



An example minibatch of size 2.

The BPR loss for each minibatch can then be reduced to:

$$\frac{1}{|U_{mini}|} \sum_{u^* \in U_{mini}} -\log\left(\sigma\left(f_\theta(u^*, v_{pos}) - f_\theta(u^*, v_{neg})\right)\right)$$

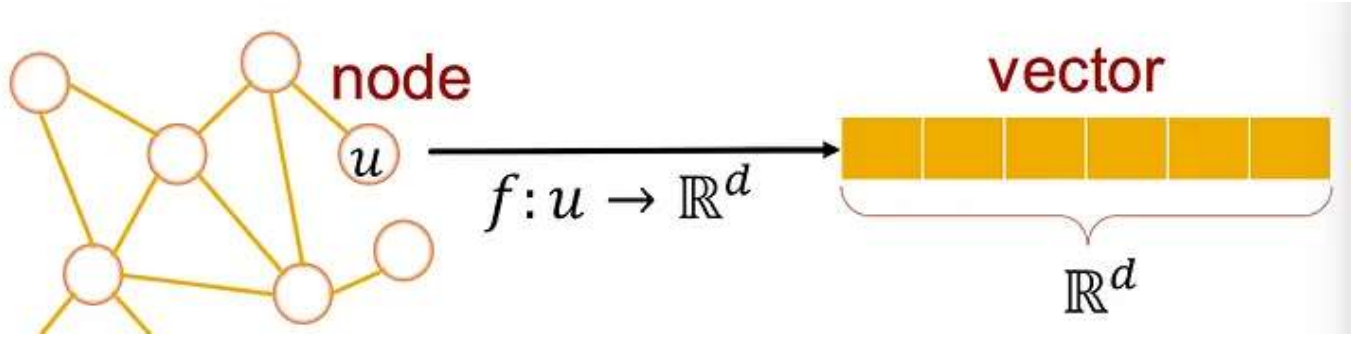
Now that we've formalized what a good movie recommender system should be able to do, we can talk more about how we can leverage GNNs.

Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) aim to learn structural and semantic patterns in input graph(s) relevant to various objective functions, similar to other deep learning domains. They work by passing messages among nodes along their edges and aggregating them for each node to update itself. This happens iteratively (hence "deep" learning) which allows individual nodes to gather information from multi-hop neighbors and infer structural features around them. This technique achieves state-of-art with its many variants and applications on social media graphs, chemical compound graphs, traffic graphs, and among many others, recommender system graphs.

As with all deep learning, we would like to have some representation of the latent features for the nodes in a graph. This is most efficiently represented as an embedding, or a vector u that is d-dimensional and consists of real values. **Node embeddings** are helpful in the domain of graph neural networks because they can encode information such as similarity between different nodes or graph structure information, which can be helpful for downstream predictions. Previously, we used shallow embeddings in the matrix factorization for CF, but now we use deep embeddings which can learn richer features about the structure of our graphs.

With node embeddings, we must have an encoder that maps nodes to embeddings, and a decoder that maps embeddings to some sort of score that is meant for our network prediction task. We decide to use deep encoders in the form of graph neural networks (GNNs) in order to learn the necessary features for our recommendation task.



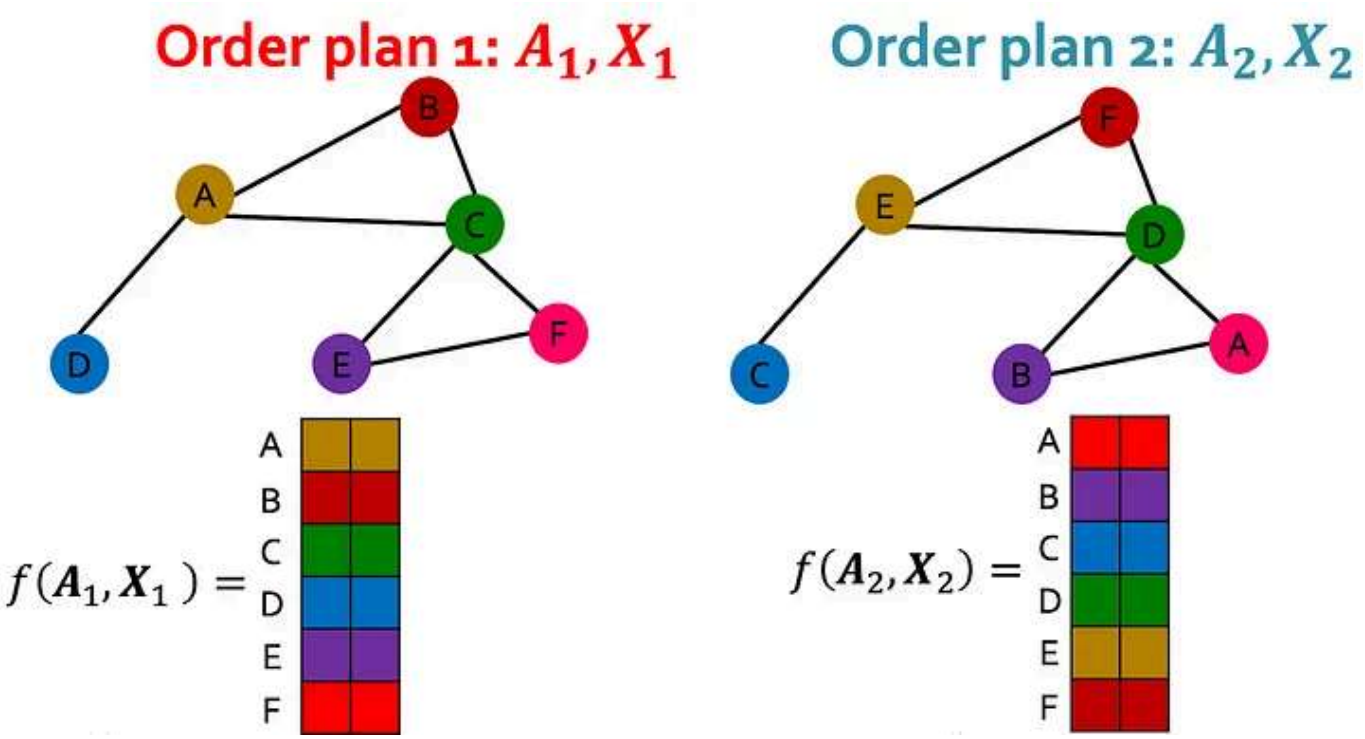
A node can be encoded into an embedding vector that is d-dimensional.

A **graph neural network**, or **GNN**, is a deep learning method that applies layers of non-linear transformations of node features by leveraging graph structure. GNNs are fundamentally similar to other deep learning approaches, still relying on loss functions, backpropagation, and stochastic gradient descent in order to optimize some training objectives.

However, GNNs must also adhere to two more requirements: 1) permutation invariance, and 2) permutation equivariance.

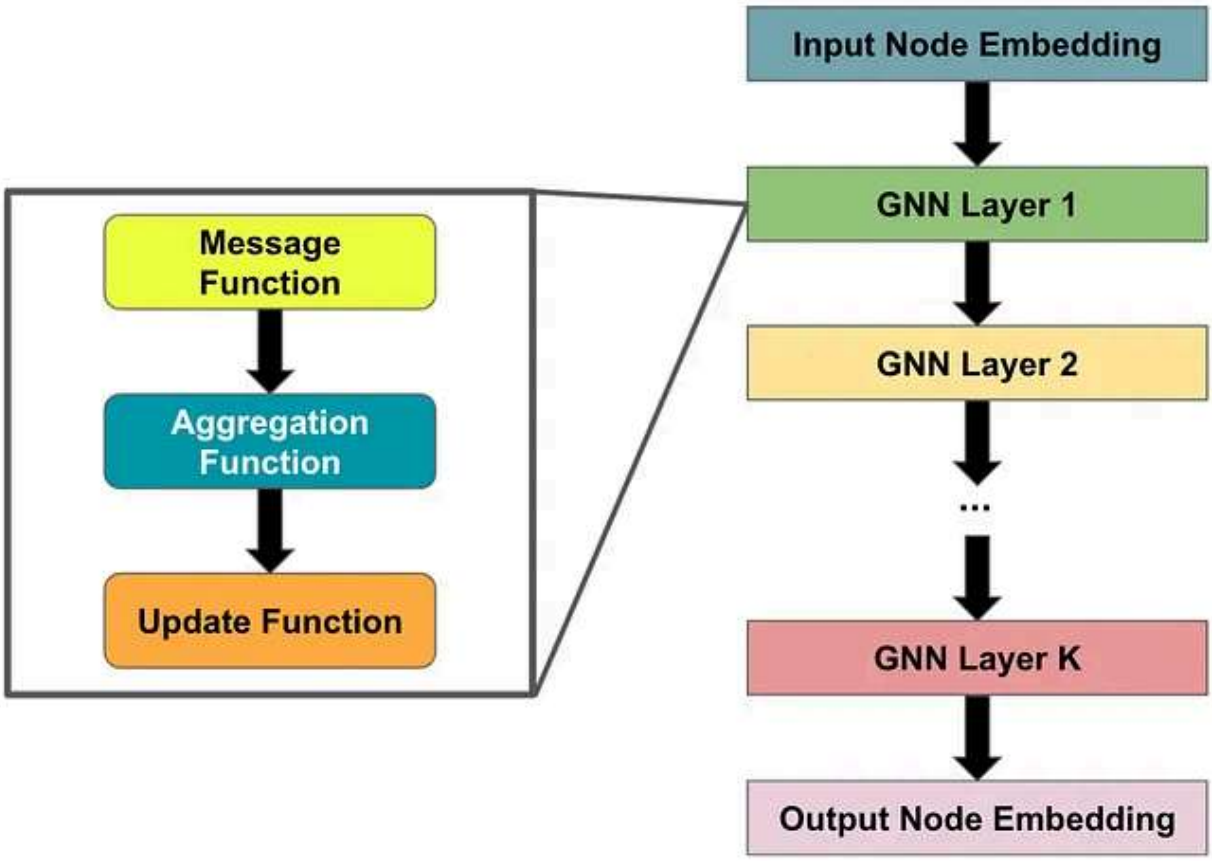
Permutation invariance means that the graph representation that the GNN produces must be the same for any two order plans, meaning that we must ensure that there is no ordering of the nodes that affects the model's output. Permutation equivariance means that for any function that maps nodes of G needs to produce the same representation for the same node, even if it is outputted at a different position, as long as the node is the same for a different ordering plan.

In the diagram below, the output of both ordering plans must be the same since they are both the same graph, and the representation of specific nodes must be the same even if they are in different positions of the output.



For any two order plans, the GNN must maintain permutation invariance and permutation equivariance.

The main idea of GNNs is to use a node's neighborhood to define a computation graph to generate the node's embeddings. As a result, we must somehow have our model learn to propagate and transform information from a node's computation graph to generate the feature representation. Thus, GNNs will aggregate information from each node's neighbors using neural networks. These GNNs are defined by stacking layers of message and aggregation functions, where each additional layer represents a node getting information from its neighbors who are an additional hop away. The inputs to the GNN at the first layer will be the input features of the node.



A high level overview of the GNN model and propagation framework.

Each layer of a GNN consists of three fundamental components: the message function, aggregation function, and update function. We will see examples of what these functions are later in this blog when we describe the models we are using for recommendation, but we will explain what they are at a high level right now.

The **message function** is meant to send information from a node's neighbors to the current node. A common way to do this is to use the hidden representation of neighbor node u, and pass it into the computation for creating the embedding of the current node v. The message function may also use information about the hidden representation of the current node v itself at the current layer to compute the next since information on the hidden representation from the previous layer may be important to retain.

The **aggregation function** is meant to combine the message information produced by each neighboring node during the message function. This function must be permutation invariant since there's no ordering that is set for our nodes. Some common aggregation functions are mean and summation.

Finally, the GNN will have an **update function** to update the representation of the node based on the output of the aggregation function. Typically, an update function would have elements such as a multi-layer perceptron, a skip connection, and/or a non-linear activation.

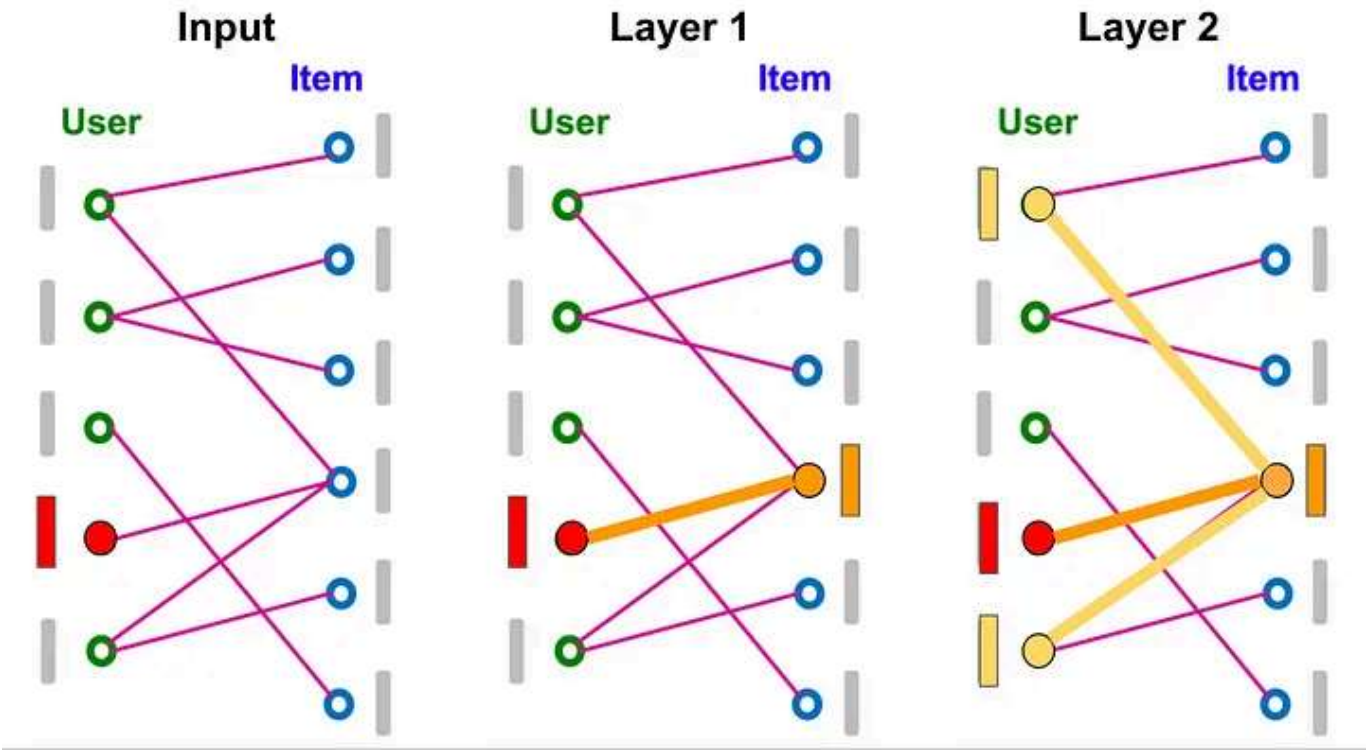
How to Use GNNs for the Recommendation Task

So how exactly can we use a GNN to create a recommender system? Well, the answer is to use collaborative filtering. As mentioned before, collaborative filtering takes the assumption that a user who shares similar interests with another user will also like an item that the other user previously liked.

Conventionally, CF is modeled using matrix factorization. As described earlier, this simple model uses $score(u,v) = f\_theta(u, v)$ as the score function to indicate how likely someone is to like the recommendation. While this model is a good first step to understanding how to develop more complex recommendation models, it does not explicitly capture enough information. Specifically, matrix factorization cannot capture graph structure information, such as the K-hop paths between two nodes, which is information that can be captured using deep neural network approaches. Let's see how we can use GNN-based models with a more complicated choice of the function f to capture this information.

Deep approaches using GNNs can explicitly capture local graph structure as well as high-order graph structure. Both of these components are important

for a recommendation system to successfully generate user and item node embeddings. The most useful property of GNNs is the message passing, which allows for the propagation of embeddings along edges of the node. As a result, for GNNs that use K layers, each node is aware of the graph structure that is K-hops away. Another way to think about this is to consider how a user node can look at the items it is interested in, find the user nodes who also express interest in the same items, and gain information from the other user nodes to update its own embedding. Similarly, an item node can look at what users are interested in and find the other items the users are interested in to create an item embedding based off of those other items. This allows users who have similar interests as well as items who have similar users interested in them to have embeddings converge towards each other and optimize the scoring function for recommendation.



An example of what information the user node highlighted in red gets from its neighbor embeddings k-hops away in each layer output of GNN propagation.

In the above figure, we demonstrate how a GNN propagates information in a helpful way. Specifically, we consider the information the user embedding in red would have at each layer of the GNN. First, the user node would be the input embedding provided to the model. Then, in Layer 1, the user node embedding gets updated with messages from the item embedding in orange. Note that during this layer, the item node in orange also learns of the user embeddings in yellow during Layer 1. Finally, in Layer 2, the user node embedding in red will then get updated with the new item embedding in orange, which also contains information about the other user embeddings in yellow. As a result, the user node in red will have gained information about the embeddings of users that are 2-hops away in a GNN that performs 2 layers of graph convolution operations. Semantically, what this means in terms of recommendations is that we have made this user's node embedding more similar to other users who share the same item interests.

In the remainder of this post, we demonstrate how to implement two popular GNN architectures, Neural Graph Collaborative Filtering (NGCF) and LightGCN, for recommender systems in PyG and compare their performance. Both architectures learn initial feature embeddings for each user and item, which are propagated according to the graph structure to prepare final embeddings. Similarly, both architectures compute the score of a user and item to be the inner product of their final embeddings. In other words, the score function is:

$$score(u, v) = f_\theta(u, v) = (e_u^{(l)})^T e_v^{(l)}$$

where l is the number of layers in the network.

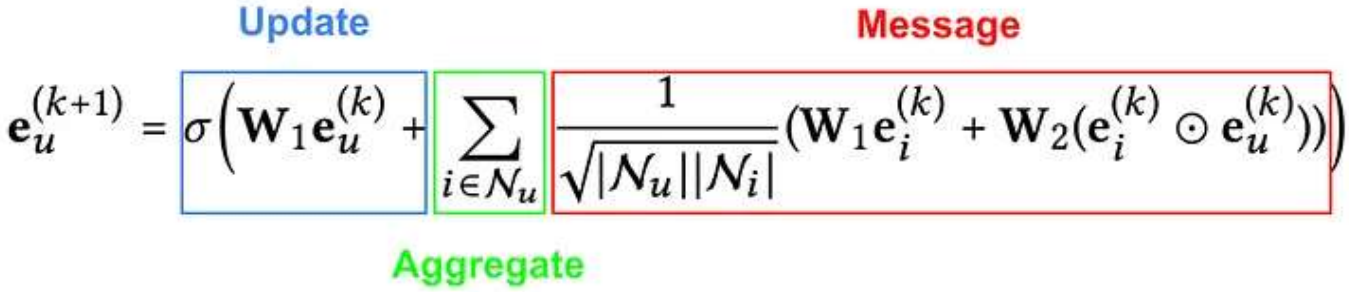
Neural Graph Collaborative Filtering (NGCF)

As part of each NGCF layer, the model applies learnable feature transformations as well as activation functions to update the embedding for that layer. The update rules for the user and item embeddings are described by the following two functions.

$$\mathbf{e}_u^{(k+1)} = \sigma\left(\mathbf{W}_1\mathbf{e}_u^{(k)} + \sum_{i \in \mathcal{N}_u} \frac{1}{\sqrt{|\mathcal{N}_u||\mathcal{N}_i|}}(\mathbf{W}_1\mathbf{e}_i^{(k)} + \mathbf{W}_2(\mathbf{e}_i^{(k)} \odot \mathbf{e}_u^{(k)}))\right)$$

$$\mathbf{e}_i^{(k+1)} = \sigma\left(\mathbf{W}_1\mathbf{e}_i^{(k)} + \sum_{u \in \mathcal{N}_i} \frac{1}{\sqrt{|\mathcal{N}_u||\mathcal{N}_i|}}(\mathbf{W}_1\mathbf{e}_u^{(k)} + \mathbf{W}_2(\mathbf{e}_u^{(k)} \odot \mathbf{e}_i^{(k)}))\right)$$

Let's break down these functions, specifically by looking at it specifically from the user embedding (*e_u*) perspective.



For the message function, each of the neighboring item nodes for a specific user will first have its features transformed by matrix *W1*. Then, we add that with the element-wise matrix multiplied embeddings between the item and the user that has had its features transformed by matrix *W2*. Finally, we also combine the message with the symmetric normalization term, which prevents the embeddings from growing in value as more graph convolution operations are performed.

The message function is implemented in PyG as follows:

```
def message(self, x_j, x_i, norm):
    return norm.view(-1, 1) * (self.lin_1(x_j) + self.lin_2(x_j *
x_i))
```

For the aggregation function, we see that the update function for the embedding performs a summation over all the neighbor item nodes for any specific user node. As a result, for aggregation we will simply sum over the messages for any specific node.

```
def aggregate(self, x, messages, index):
    out = torch_scatter.scatter(messages, index, self.node_dim,
reduce="sum")
    return out
```

Finally, the NGCF model performs an update rule that first adds the embedding for the node at the current layer with its features transferred by matrix *W1*. Then, the embedding will be updated with a nonlinear activation function. In PyG, the forward function will apply the update step as follows:

```
def forward(self, x, edge_index):
    norm = compute_normalization(x, edge_index)
    out = self.propagate(edge_index, x=(x, x), norm=norm)
    # Update step
    out += self.lin_1(x)
    out = F.dropout(out, self.dropout, self.training)
    return F.leaky_relu(out)
```

Putting these message and forward functions together we can create a PyG `MessagePassing` class to describe one NGCF layer:

```
class NGCFConv(MessagePassing):
    def __init__(self, latent_dim, dropout, bias=True, **kwargs):
        super(NGCFConv, self).__init__(aggr='add', **kwargs)
        self.dropout = dropout
        self.lin_1 = nn.Linear(latent_dim, latent_dim, bias=bias)
        self.lin_2 = nn.Linear(latent_dim, latent_dim, bias=bias)
        self.init_parameters()
    def init_parameters(self):
        nn.init.xavier_uniform_(self.lin_1.weight)
        nn.init.xavier_uniform_(self.lin_2.weight)
    def forward(self, x, edge_index):
        # Compute normalization
        from_, to_ = edge_index
        deg = degree(to_, x.size(0), dtype=x.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        deg_inv_sqrt[deg_inv_sqrt == float('inf')] = 0
        norm = deg_inv_sqrt[from_] * deg_inv_sqrt[to_]
        # Start propagating messages
        out = self.propagate(edge_index, x=(x, x), norm=norm)
        out += self.lin_1(x)
        out = F.dropout(out, self.dropout, self.training)
        return F.leaky_relu(out)
    def message(self, x_j, x_i, norm):
        return norm.view(-1, 1) * (self.lin_1(x_j) + self.lin_2(x_j
* x_i))
```

We can use that NGCF layer to create a PyTorch model that implements a multi-layer NGCF. During the forward function, we will simply keep track of the embeddings produced at each layer and concatenate them before outputting the final embeddings:

```
def forward(self, edge_index):
    emb0 = self.embedding.weight
    embs = [emb0]
    emb = emb0
    for conv in self.convs:
        emb = conv(x=emb, edge_index=edge_index)
        embs.append(emb)
        out = torch.cat(embs, dim=-1)
        return out
```

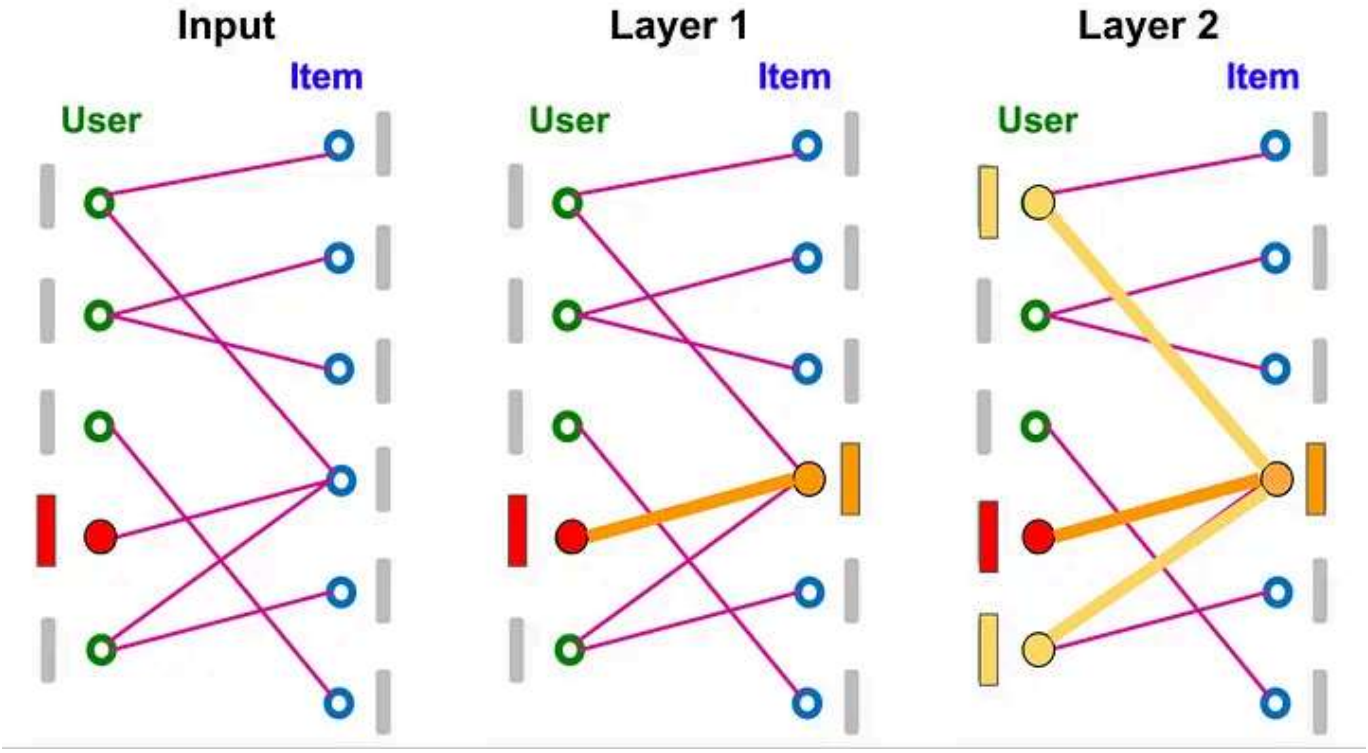
Using these final embeddings, we can implement the BPR loss function for the user, positive item, and negative item embeddings from a single minibatch.

```
def bpr_loss(users_emb, pos_emb, neg_emb):
    pos_scores = torch.mul(users_emb, pos_emb).sum(dim=1)
    neg_scores = torch.mul(users_emb, neg_emb).sum(dim=1)
    bpr_loss = torch.mean(F.softplus(neg_scores - pos_scores))
    return bpr_loss
```

LightGCN

LightGCN is a neural graph model that focuses on the most important component of the network: neighborhood aggregation. Specifically, LightGCN only uses the initial node embeddings of the users and items in the graph as the learnable parameters of the network. Unlike NGCF, LightGCN gets rid of the GNN parameters for each layer and avoids using feature transformations or nonlinear activation functions to reduce the complexity of the model. The main rationale for removing these complexities is because shallow learnable embeddings for the users and items are already expressive enough.

Moreover, since the main purpose of collaborative filtering schemes are to find other similar users based on their preferences, simply diffusing node embeddings along the graph by doing neighborhood aggregation is enough to capture the information needed to create suitable embeddings for predicting if a user will like an item. A simple way to visualize this is in the following figure, where users are on the left and items are on the right, and we see that the embedding of the highlighted node relies on nodes for K-hops away, depending on how many layers are used in the LightGCN model.



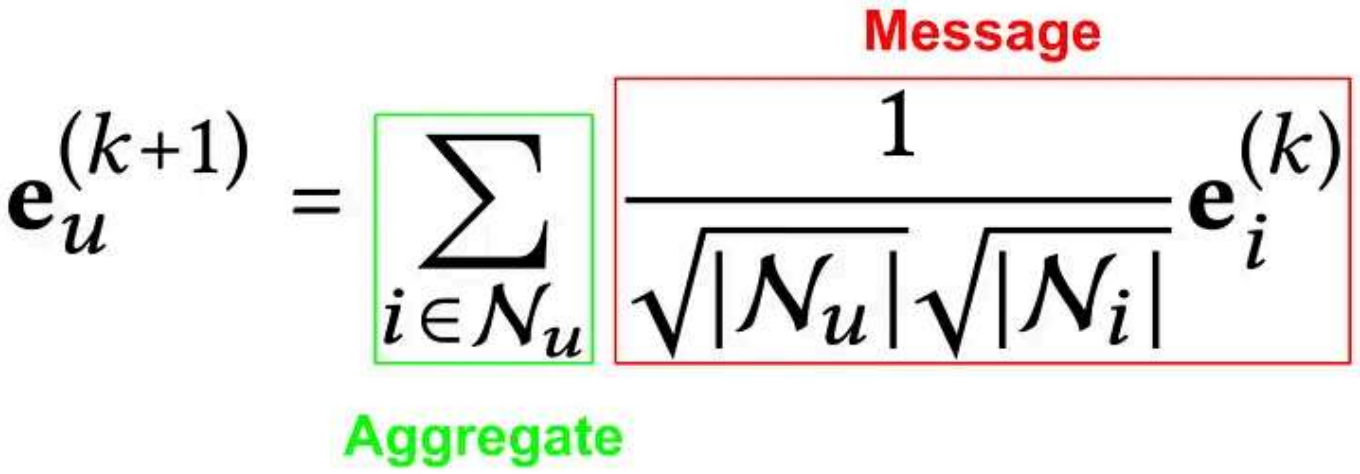
Now we will go over the various components of the LightGCN model and its implementation in PyG.

The new update rules for the user and item embeddings are now described by the following two functions.

$$\mathbf{e}_u^{(k+1)} = \sum_{i \in \mathcal{N}_u} \frac{1}{\sqrt{|\mathcal{N}_u|}\sqrt{|\mathcal{N}_i|}} \mathbf{e}_i^{(k)},$$

$$\mathbf{e}_i^{(k+1)} = \sum_{u \in \mathcal{N}_i} \frac{1}{\sqrt{|\mathcal{N}_i|}\sqrt{|\mathcal{N}_u|}} \mathbf{e}_u^{(k)}.$$

Let's break down these functions, specifically by looking at it specifically from the user embedding (*e_u*) perspective.



For the message function, we note that each user node is simply gathering all of the information from the neighbor nodes, which are the item nodes. As a result, the message function is to simply pass the embedding of the neighboring item nodes directly. We also decide to combine the message with the symmetric normalization term, which prevents the embeddings from growing in value as more graph convolution operations are performed.

The message function is implemented in PyG as follows:

```
def message(self, x_j, norm):
    return norm.view(-1, 1) * x_j
```

For the aggregation function, we see that the update function for the embedding performs a summation over all the neighbor item nodes for any specific user node. As a result, for aggregation we will simply sum over the messages for any specific node.

```
def aggregate(self, x, messages, index):
```

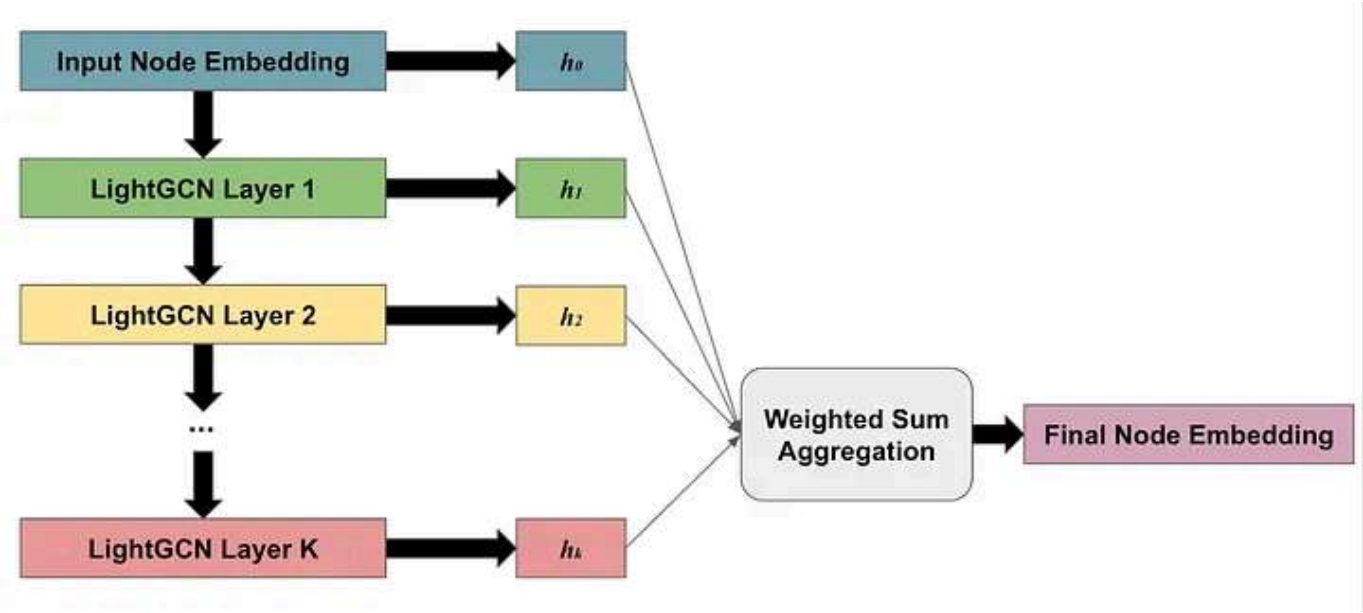
```
        out = torch_scatter.scatter(messages, index, self.node_dim,
    reduce="sum")

        return out
```

Finally, the LightGCN model combines the intermediate representations of each layer of the model in order to get the final embedding output for users and items. The model does so using a weighted sum in the following equations:

$$\mathbf{e}_u = \sum_{k=0}^{K} \alpha_k \mathbf{e}_u^{(k)}; \quad \mathbf{e}_i = \sum_{k=0}^{K} \alpha_k \mathbf{e}_i^{(k)};$$

The term ***alpha_k*** is commonly set uniformly as *1/(K+1),* which means that the summation is simply the mean of the output embeddings of all ***K*** layers as well as the initial input embedding which is learned. The reason why the intermediate embeddings are summed are as follows:

1. Prevent over-smoothing, as adding more layers will cause all embeddings to converge towards each other, so we also add embedding outputs from earlier layers
2. Capture different semantics, such as smoothing between users and their interacted items in the first layer, smoothing users with other similar users based on their interacted items, etc.



A high level outline of the LightGCN model and aggregation of outputs.

Putting these message and forward functions together we can create a PyG `MessagePassing` class to describe one LightGCN layer:

```
class LightGCNConv(MessagePassing):
      def __init__(self, **kwargs):
            super().__init__(aggr='add')
      def forward(self, x, edge_index):
            # Compute normalization
            from_, to_ = edge_index
            deg = degree(to_, x.size(0), dtype=x.dtype)
            deg_inv_sqrt = deg.pow(-0.5)
            deg_inv_sqrt[deg_inv_sqrt == float('inf')] = 0
            norm = deg_inv_sqrt[from_] * deg_inv_sqrt[to_]
            # Start propagating messages
            return self.propagate(edge_index, x=x, norm=norm)
      def message(self, x_j, norm):
           return norm.view(-1, 1) * x_j
```

We can use that LightGCN layer to create a PyTorch `nn.module` that implements a multi-layer LightGCN. During the forward function, we will simply keep track of the embeddings produced at each layer and perform a mean-weighted sum to output the final embeddings:

```
def forward(self, edge_index):
        emb = self.embedding.weight
        embs = [self.embedding.weight]
        for conv in self.convs:
                emb = conv(x=emb, edge_index=edge_index)
                embs.append(emb)
        # perform weighted sum on output of all layers to yield final
embedding
        embs = torch.stack(embs, dim=0)
        out = torch.mean(embs, dim=0)
        return out
```

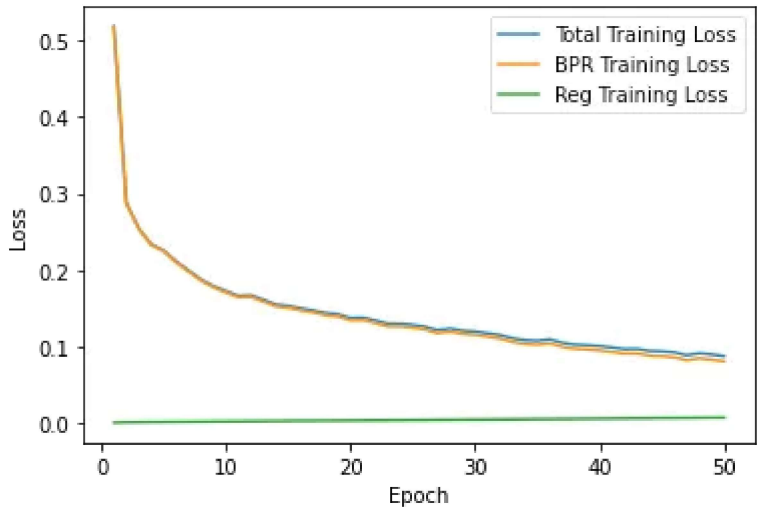
Despite the LightGCN embeddings having a different size than those produced by NGCF (since LightGCN takes a weighted-sum of intermediate-layer embeddings rather than concatenating them), we can use the same implementation of BPR loss for a single minibatch.

Results & Conclusion

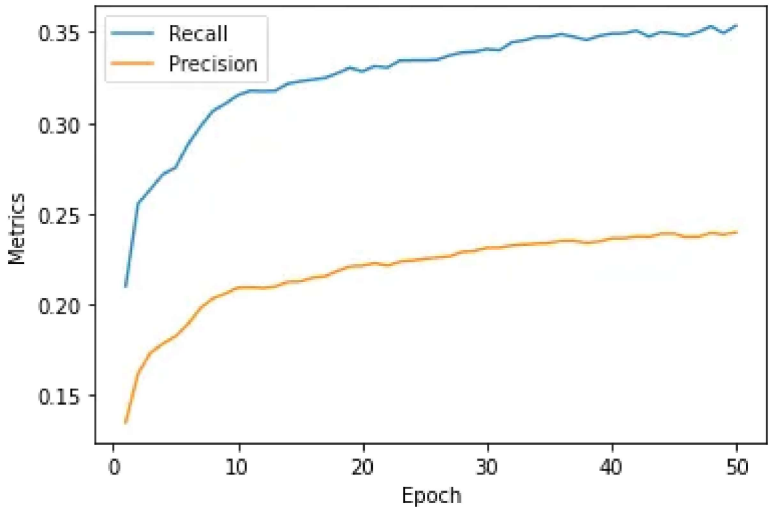
Now that we've implemented both LightGCN and NGCF in PyG, let's see which one of them performs better on the MovieLens 100k dataset!

We split the dataset into an 80/20 train-test split and train both models using the minibatching procedure described above. The held-out edges in the test

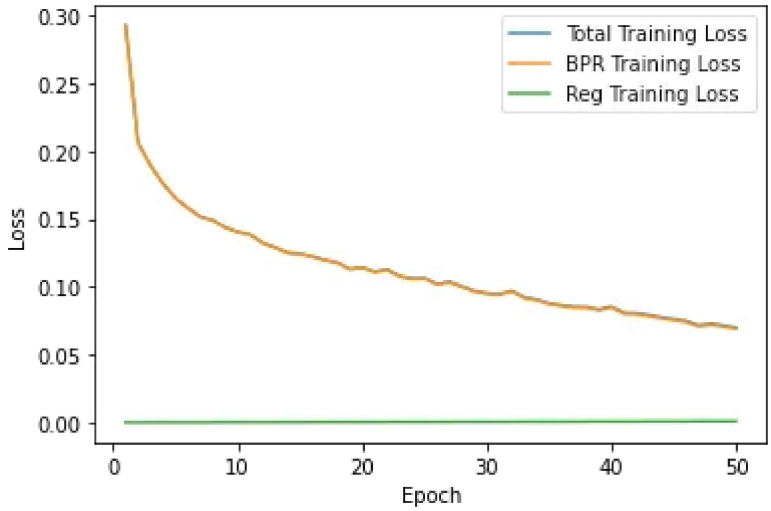
set are used to evaluate precision@K and recall@K after each training epoch.



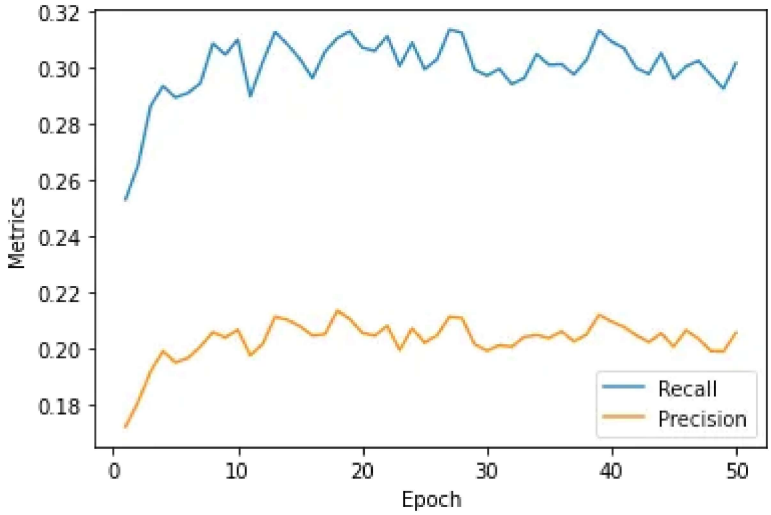
LightGCN loss curves per training epoch.



LightGCN precision@20 and recall@20 per training epoch.



NGCF loss curves per training epoch.



NGCF precision@20 and recall@20 per training epoch.

We can make a couple of observations comparing the results and performance of each of these two models:

- The precision@20 and recall@20 curves for NGCF are significantly less smooth than those of LightGCN.
- Even while the loss for NGCF continues to go down, the precision@20 and recall@20 curves plateau and oscillate.

- LightGCN performs better than NGCF with respect to both metrics. Specifically, LightGCN achieves a recall@20 of 0.3535 compared to 0.3136 for NGCF, a 13% increase. It also achieves a precision@20 of 0.2397 compared to 0.2134, a 12% increase.

That means we are able to replicate one of the key results of the LightGCN paper on a new dataset. Across three datasets, the authors of LightGCN found that it outperformed NGCF by similar margins to our experiments on MovieLens 100k here.

Coming from a more general deep learning background, this is one of the counterintuitive results to come out of graph machine learning. Why would the ostensibly less expressive model, with fewer learnable parameters perform better?

One hypothesis is that the feature transformation and nonlinear activation performed by NGCF increases training difficulty while not significantly improving the model's learning ability. The majority of the learnable parameters in both models are inside of the initial user and item embeddings, not in the weight matrices. This means that LightGCN is nearly as expressive as NGCF if you count parameters alone. LightGCN doubles-down on the power of harnessing the graph structure directly in order to achieve higher performance

too much, as LightGCN actually performs better while not being as complicated as NGCF. Or, you could just take the simple path next time and let the algorithm pick the movie for your next Netflix watch party ;)

Paper References

1. Harper, F. Maxwell, and Konstan, Joseph A. “The MovieLensDatasets: History and Context.” ACM Transactions on Interactive Intelligence Systems (TiiS) 5, 4. 2015.
2. He, Xiangnan, et al. “LightGCN: Simplifying and powering graph convolution network for recommendation.” Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval. 2020.
3. Wang, Xiang, et al. “Neural graph collaborative filtering.” Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval. 2019.

Image References

Images describing node embeddings and order plans were adapted from the CS224W coursework and lecture slides. Image highlighting node embedding perceptive field at K-hops was modified from an image in the lecture slides. All other images and figures are novel generations from this project.

Cs224w | Graph Neural Networks | Stanford | Machine Learning | Recommendation System

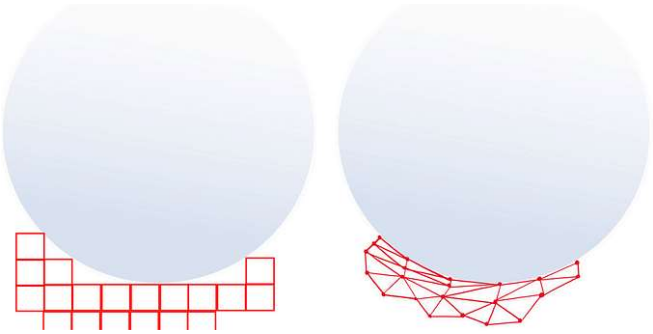
Written by Derrick Li

Follow

57 Followers · Writer for Stanford CS224W: Machine Learning with Graphs

Stanford CS '21

More from Derrick Li and Stanford CS224W: Machine Learning with Graphs



Rayan K... in Stanford CS224W: Machine Learning...

Learning Mesh-Based Flow Simulations on Graph Networks

Traditional deep learning methods are not able to model intricate mesh-based flow...

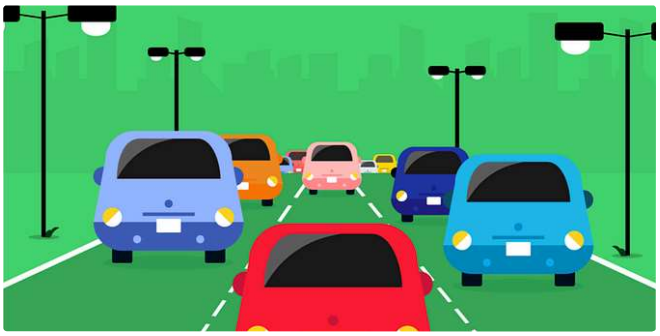
Feb 8, 2022 116 2

Michael ... in Stanford CS224W: Machine Learnin...

Tackling the Traveling Salesman Problem with Graph Neural...

Do you ever feel like there aren't enough hours in the day to get everything done? You...

May 15, 2023 103

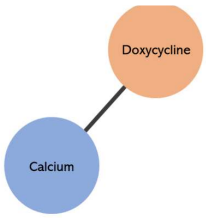


Amelia Wo... in Stanford CS224W: Machine Learn...

Predicting Los Angeles Traffic with Graph Neural Networks

By Julie Wang, Amelia Woodward, Tracy Cai as part of the Stanford CS224W course...

Jan 15, 2022 176 3



Tanish ... in Stanford CS224W: Machine Learning ...

Online Link Prediction with Graph Neural Networks

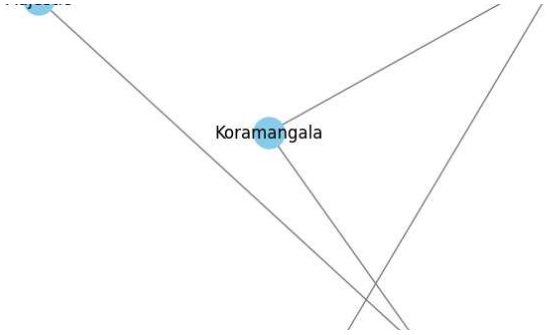
We'll be exploring the inductive power of GNNs on online link prediction by using the...

Jan 25, 2022 163 3

See all from Derrick Li

See all from Stanford CS224W: Machine Learning with Graphs

Recommended from Medium



shashank Jain in AI Mind

Spatiotemporal Graph Neural Networks (STGNNs) for Traffic...

In the age of smart cities, managing and predicting traffic effectively has become...

Sep 9 130

Yu-Cheng Tsai in Towards Data Science

A Step-by-Step Guide to Build a Graph Learning System for a Movi...

Built with PyTorch Geometric and using MovieLens DataSet

Sep 11 379 2

Lists

Predictive Modeling w/ Python

20 stories · 1581 saves

Practical Guides to Machine Learning

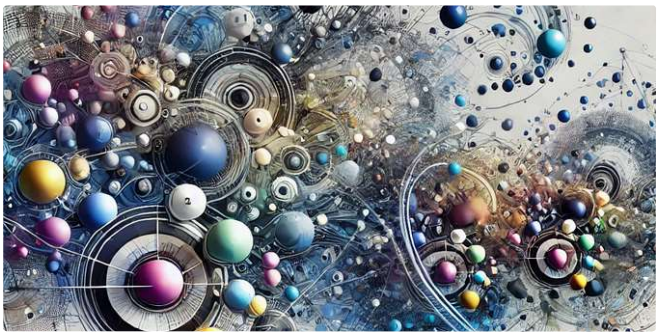
10 stories · 1918 saves

Natural Language Processing

1741 stories · 1335 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 473 saves



Hennie de Harder in Towards Data Science

Graph Neural Networks Part 1. Graph Convolutional Networks...

Node classification with Graph Convolutional Networks

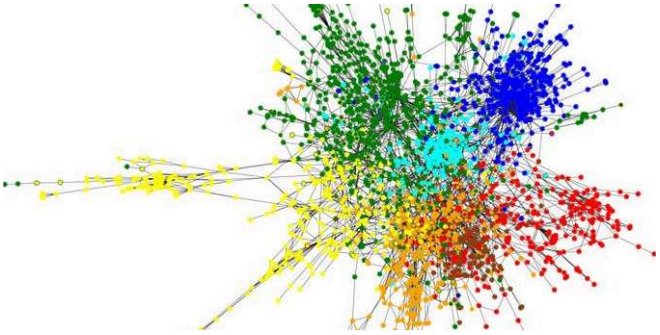
6d ago 324

Sahil Sheikh

Link Prediction in GNNs Made Easy- Deep Graph Library (DGL)

I have worked on a few GNN projects before in which I have used Neo4j's Graph Data Scien...

May 11 9 1



Abin Varghese

Dive into Graph Neural Networks with PyTorch: A Simple Guide

Hey there! Today, I'm excited to walk you through a cool project I've been working on...

Jun 19 28

Lem Apperson

Getting Started with Unreal Engine

Introduction to Unreal Engine Projects: Structure and Components

Apr 12

See more recommendations