Project Report:
LightGCN for Movie Recommendation

Le Hoang Long
Hanoi University of Science and Technology (HUST)
No. 1 Dai Co Viet, Hai Ba Trung, Ha Noi, Viet Nam
long.lh232099m@sis.hust.edu.vn, hoanglong1712@gmail.com

Abstract

Movie recommendation is a fascinating field. For instance, every time we open YouTube, we're greeted with a list of new videos, some familiar and others unfamiliar. This process is known as movie recommendation. A recommender system generates a tailored list of user suggestions based on two main factors: 1) their interactions with other users and 2) their preferences for specific items. The technique of using user-user and user-item interactions to predict future recommendations is called Collaborative Filtering (CF).

Collaborative Filtering (CF) is one of recommender systems' most widely studied methods. It operates on the principle that if one user enjoys an item that another user with similar preferences liked in the past, the first user is likely to appreciate that item as well. CF utilizes historical interactions between users and items to forecast potential future preferences.

In this paper, we will examine a cutting-edge model based on Graph Neural Networks (GNN), specifically LightGCN as a movie recommender system using the MovieLens 1M dataset.

Figure 1: Like, Dislike & Unrated

1 MovieLens Dataset and Graph Neural Networks in Recommender Systems

1.1 MovieLens Dataset

MovieLens is an excellent dataset for training movie recommendation systems. The MovieLens 1M dataset is particularly well-suited for smaller projects, containing 1 million movie ratings, 4,000 movies, and 6,000 users.

Moreover, the MovieLens datasets come in various sizes, such as 100K, 10M, and 25M, providing us the flexibility to choose the dataset that best aligns with our computing capabilities and training objectives.

The dataset is well-maintained and thoroughly validated, facilitating quick training. It has been referenced in numerous studies. All features, including those related to users and movies, are easy

to interpret. Movie features include titles and genres, while user features encompass gender, age, occupation, and zip code. These features are often thought to be potentially linked to people's preferences.

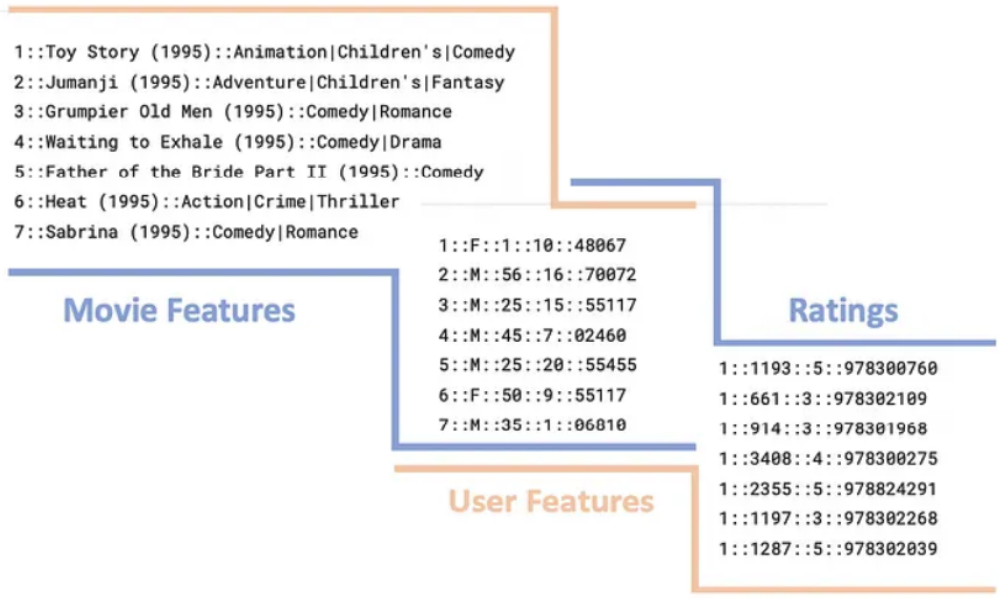


Figure 2: Movie Features - User features - Ratings

1.2 Graph Neural Network

The movie recommendation model is a sub-task of the linked-level prediction task, which involves predicting new, missing, or unknown connections based on existing ones. During testing, node pairs that lack established links are evaluated and ranked, with the top K pairs being predicted.

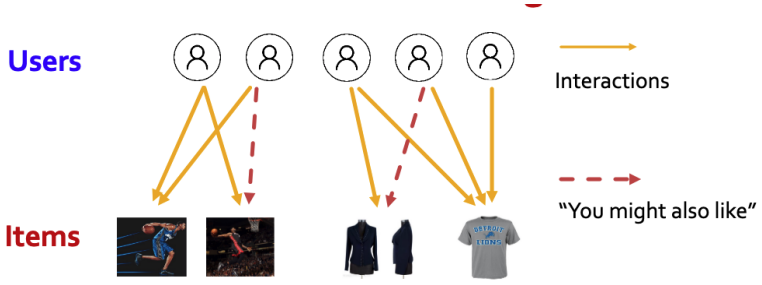


Figure 3: Recommend items users

In a Graph Neural Network (GNN), entities are represented as nodes, while relationships between them are depicted as edges. The graph can be enhanced by adding node and edge features for more information. GNNs generate embeddings by considering the structure of the graph. Specifically, they use the embeddings of a central node's neighbors—defined as nodes within a certain distance or "hop" from the central node—to update its own embedding. This approach enables the GNN to produce similar embeddings for nodes that are closely related, both locally and from a broader perspective.

It requires a mapping function f that maps nodes to d-dimensional embeddings such that similar nodes in the graph are embedded close together

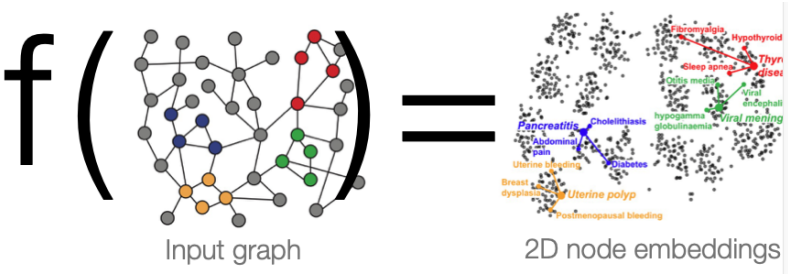


Figure 4: Mapping function f

Our objective is to establish the similarity between two nodes within the embedding space.

$$similarity(u, v) \approx z_v^T z_u$$

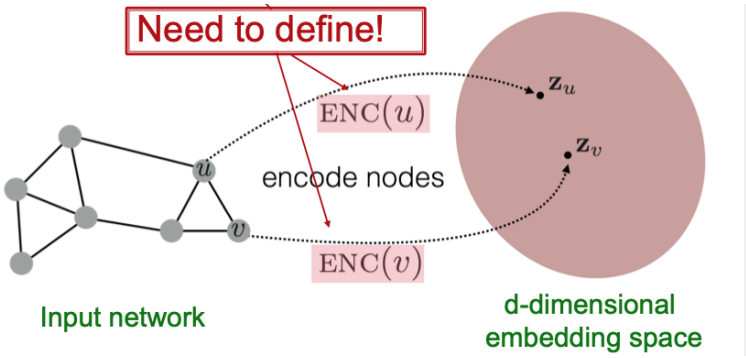


Figure 5: $ENC(v)$ = multiple layers of non-linear transformations based on graph structure

During training, a node transmits "messages" as vector embeddings to its neighboring nodes and receives messages from them as well. Each node aggregates these messages using a permutation equivariant function, ensuring that the output remains unchanged regardless of input order. Common aggregation methods include summation, mean, and maximum. This process, known as neighborhood aggregation, allows the aggregated output to be transformed and used to update the node's embedding.

Key idea: Generate node embeddings based on local network neighborhoods

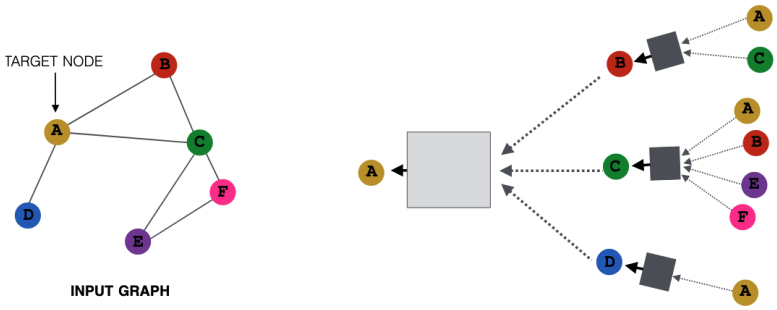


Figure 6: Node Embedding Generation

Intuition: Nodes aggregate information from their neighbors using neural networks

The updated embedding then facilitates information propagation to the next layer in the GNN training process. The underlying idea is that by passing messages through multiple hops in the neighborhood, each node learns an embedding that reflects its relationships with its neighbors.

Assume we have graph G

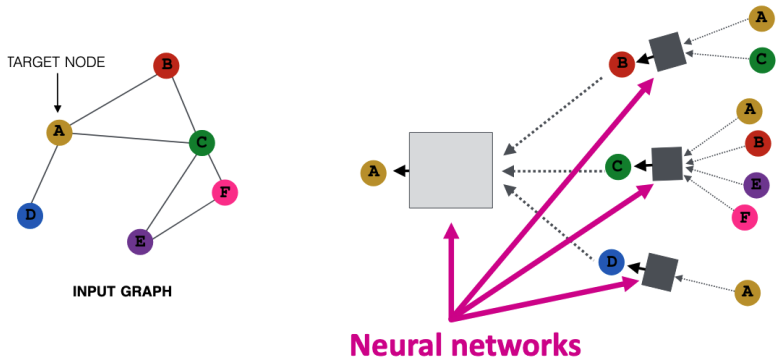


Figure 7: Neural network

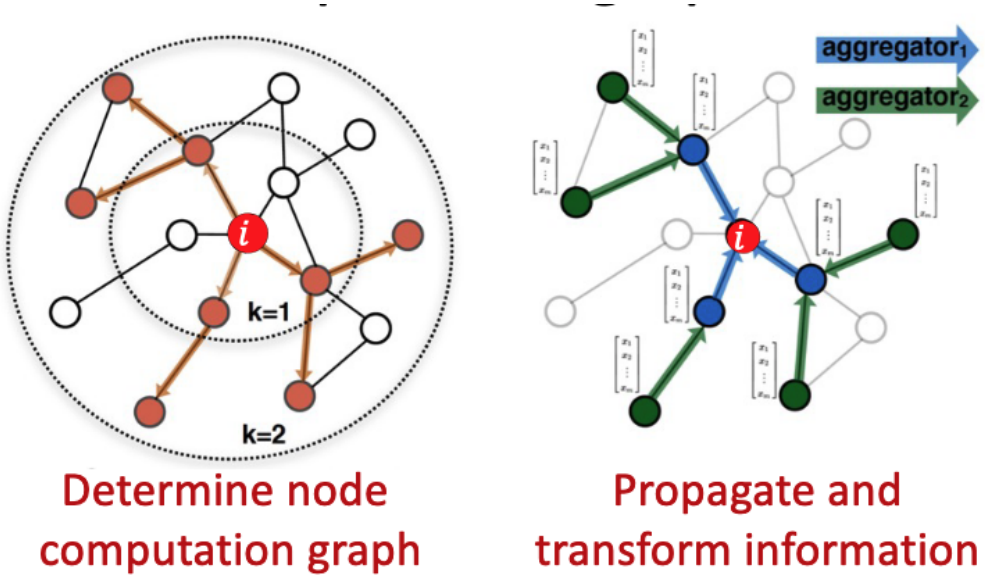


Figure 8: Computation graph

- V is the vertex set
- A is the adjacency matrix (assume binary)
- $X \in \mathbb{R}^{|V| \times m}$ is a matrix of node features
- v: a node in V; N(v): the set of neighbors of v.

Model can be of arbitrary depth:

- Nodes have embeddings at each layer
- Layer-0 embedding of node v is its input feature, x_v
- Layer-k embedding gets information from nodes that are k hops away

Model parameters

$$h_v^0 = x_v$$

$$h_v^{(k+1)} = \sigma(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(u)|} + B_k h_v^{(k)}), \forall k \in \{0 \dots K-1\}$$

$$z_v = h_v^{(K)}$$

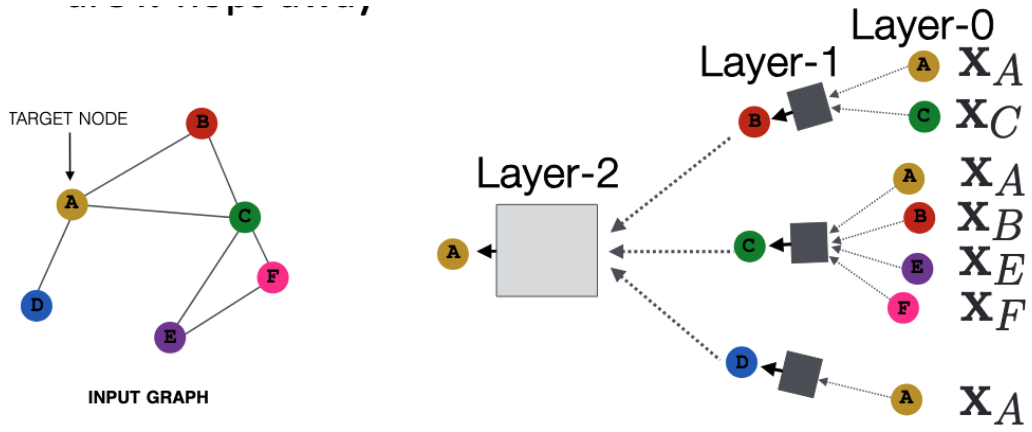


Figure 9: Multiple layers

where

- h_v^k: the hidden representation of node v at layer k, the embedding of v at layer k
- W_k: weight matrix for neighborhood aggregation
- B_k: weight matrix for transforming hidden vector of self
- z_v: final node embedding, embedding after K layers of neighborhood aggregation
- W_k and B_k are trainable weight matrices
- $h_v^0 = x_v$: Initial 0-th layer embeddings are equal to node features
- σ: non-linearity (e.g., ReLU)
- $\sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(u)|}$: average of neighbor's previous layer embeddings
- K: total number of layers

Many aggregations can be performed efficiently by (sparse) matrix operations.
Let $H^(k) = [h_1^{(k)}..h_{|V|}^{(k)}]^T$
Then $\sum_{u \in N(v)} h_u^{(k)} = A_{v,:}H^{(k)}$
Let D be diagonal matrix where $D_{v,v} = Deg(v) = |N(v)|$.
The inverse of D: D^{-1} is also diagonal. $D_{v,v}^{-1} = 1/|N(v)|$
Therefore,

$$\sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(u)|} \Rightarrow H^{(k+1)} = D^{-1}AH^{(k)}$$

Re-writing update function in matrix form:

$$H^{(k+1)} = \sigma(\tilde{A}H^{(k)}W_k^T + H^{(k)}B_k^T)$$

where $\tilde{A} = D^{-1}A$

- Red: neighborhood agregation
- Blue: self transformation

2 LightGCN

Of all GNN implementations, LightGCN stands out for achieving state-of-the-art empirical performance in recommendation benchmarks. We will develop a LightGCN model for movie recommendations based on the paper titled "LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation" https://arxiv.org/pdf/2002.02126.

We observe that the two most prevalent features in GCNs—feature transformation and nonlinear activation—have minimal impact on collaborative filtering performance. Their inclusion complicates training and can hinder recommendation effectiveness. LightGCN focuses solely on the essential aspect of GCNs: neighborhood aggregation. It learns user and item embeddings by linearly propagating them through the user-item interaction graph, ultimately using the weighted sum of embeddings from all layers as the final representation. This straightforward, linear approach simplifies implementation and training, resulting in significant improvements of around 16.0% relative improvement on average over Neural Graph Collaborative Filtering (NGCF), a leading GCN-based recommender model, all within the same experimental framework.

The core concept of Graph Convolutional Networks (GCNs) is to learn node representations by smoothing features across the graph. This is accomplished through iterative graph convolution, where the features of neighboring nodes are aggregated to create a new representation for a target node. Such aggregation can be abstracted as:

$$e_u^{k+1} = AGG(e_u^k, \{e_i^k : i \in N_u\})$$

The AGG is an aggregation function that considers the k-th layer's representation of the target node and its neighbor nodes.

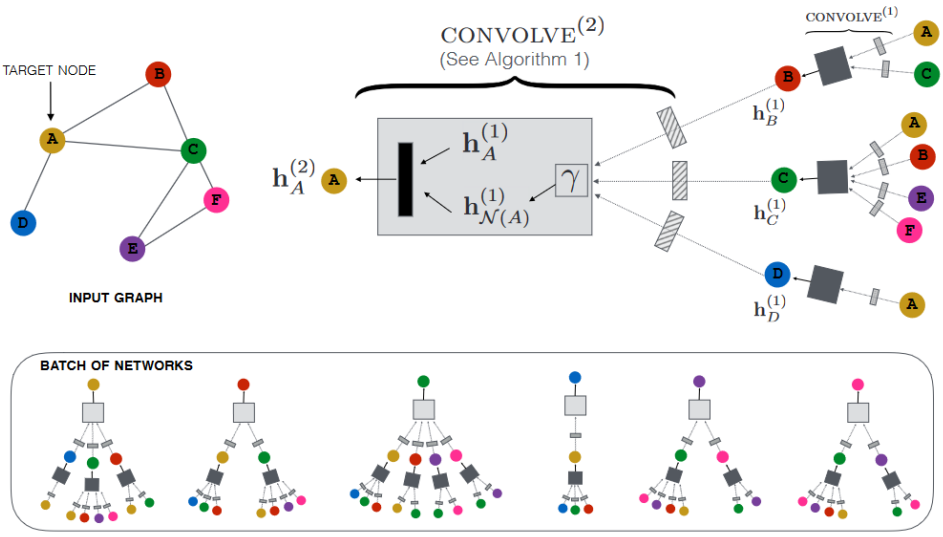


Figure 10: GCN

The fundamental concept of Graph Convolutional Networks (GCNs) is to iteratively gather feature information from local graph neighborhoods using neural networks. Each "convolution" operation processes and aggregates feature data from a node's immediate one-hop neighborhood. By stacking multiple convolution layers, information can be transmitted across broader areas of the graph. In contrast to traditional content-based deep models, GCNs utilize both content information and the underlying graph structure.

The above model architecture uses depth-2 convolutions (best viewed in color). Left: A small example input graph. Right: The 2-layer neural network that computes the embedding $h_A^{(2)}$ of node A using the previous-layer representation, $h_A^{(1)}$, of node A and that of its neighborhood $N(A)$ (nodes B, C, D).

2.1 Matrix Form

Let the user-item interaction matrix be $R \in R^{M \times N}$ where M and N denote the number of users and items, respectively, and each entry R_{ui} is 1 if u has interacted with item i otherwise 0. We then obtain the adjacency matrix of the user-item graph as

$$A = \begin{pmatrix} 0 & R \\ R^T & 0 \end{pmatrix}$$

Let the 0-th layer embedding matrix be $E^0 \in R^{M+N} \times T$, where T is the embedding size. Then we can obtain the matrix equivalent form of LGC as:

$$E^{(k+1)} = (D^{-\frac{1}{2}} A D^{-\frac{1}{2}}) E^{(k)}$$

where D is $(M+N) \times (M+N)$ diagonal matrix,in which each entry D_{ii} denotes the number of nonzero entries in the i-th row vector of the adjacency matrix A (also named as degree matrix).

We randomly initialize an embedding matrix $E^{(0)}$ for users and movies, with a shape of [# of users + # of movies, 64], where 64 represents the embedding size

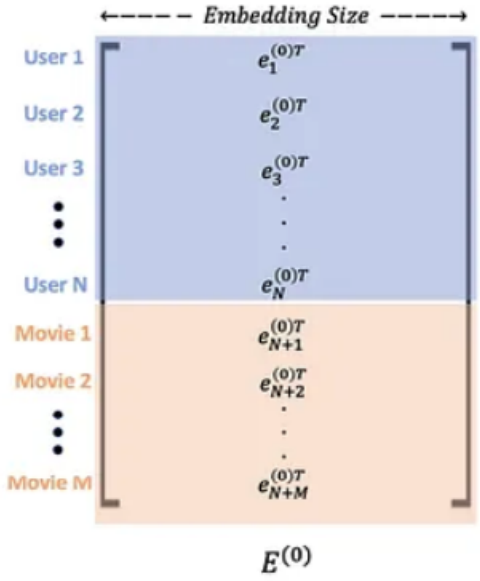


Figure 11: Embedding matrix $E^{(0)}$

Lastly, we get the final embedding matrix used for model prediction as:

$$E = \alpha_0 E^{(0)} + \alpha_1 E^{(1)} + \cdots + \alpha_k E^{(K)} = \alpha_0 E^{(0)} + \alpha_1 \tilde{A} E^{(0)} + \cdots + \alpha_k \tilde{A}^K E^{(0)}$$

where $\tilde{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ is the symmetrically normalized matrix.

2.2 Light Graph Convolution (LGC)

Like general GNNs, when applying LightGCN to the MovieLens dataset, we represent user-movie relationships as a bipartite graph. In this setup, users and movies are treated as two types of nodes on opposite sides, with edges between them signifying user-movie interactions. Our goal is to predict whether a user likes a movie, so we define the edges in the bipartite graph as follows: an edge exists if a user rates a movie 3 or higher, while no edge is present if a user rates it below 3 or has not rated it at all.

In LightGCN, we apply a simple weighted sum aggregator and abandon the use of feature transformation and nonlinear activation to compute updated embedding as the weighted sum of embeddings from all its neighboring items (movies) of each layer. The propagation rule is defined as:

$$e_u^{(k+1)} = \sum_{i \in N_u} \frac{1}{\sqrt{|N_u|}\sqrt{|N_i|}} e_i^{(k)}$$

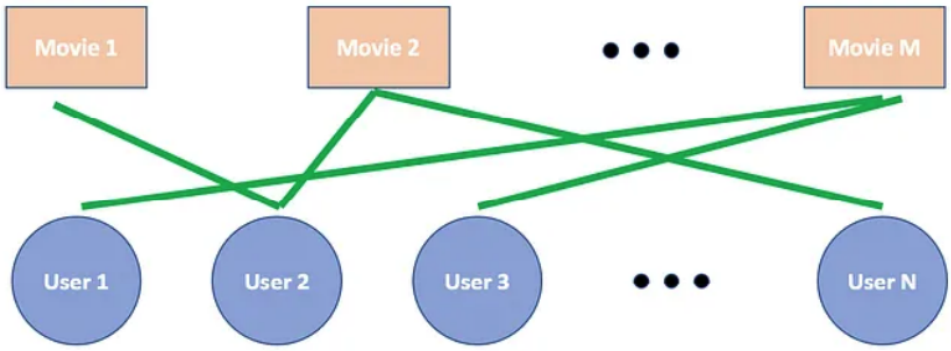


Figure 12: User-Movie bipartite graph

$$e_i^{(k+1)} = \sum_{u \in N_i} \frac{1}{\sqrt{|N_u|}\sqrt{|N_i|}} e_u^{(k)}$$

$\frac{1}{\sqrt{|N_u|}\sqrt{|N_i|}}$ is a symmetric normalization term, it helps to avoid the scale of embeddings increasing with graph convolution operations, $e_u^{(k)}$ and $e_i^{(k)}$ are the user and item (movie) node embeddings at the k-th layer. $|N_u|$ and $|N_i|$ are the user and item nodes' number of neighbors.

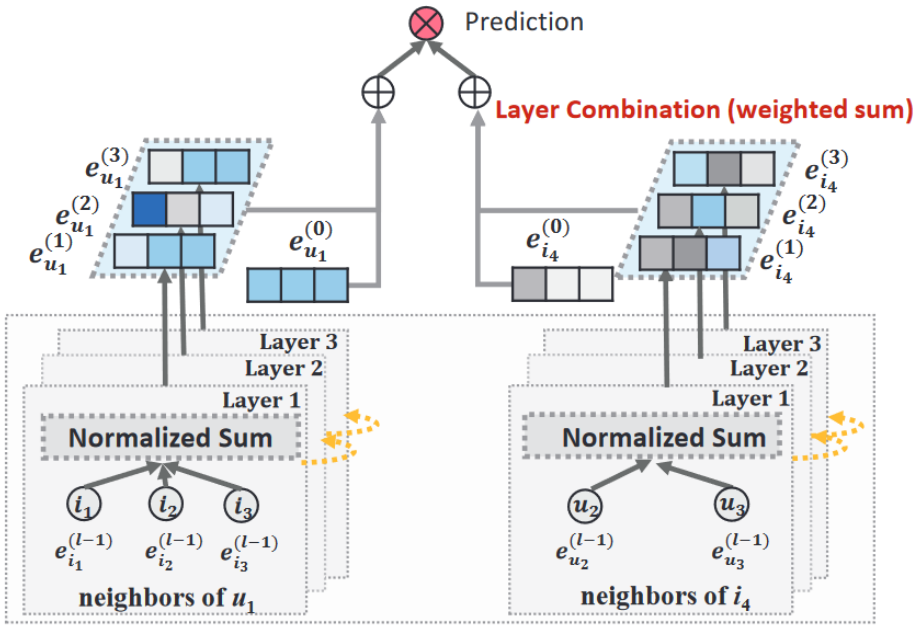


Figure 13: LightGCN

In LightGCN, only the normalized sum of neighbor embeddings is performed towards the next layer; other operations like self-connection, feature transformation, and nonlinear activation are all removed, which largely simplifies GCNs. In Layer Combination, we sum over the embeddings at each layer to obtain the final representations.

After K iterations over all the nodes, we derive the K-th layer embeddings, $E^{(K)}$.

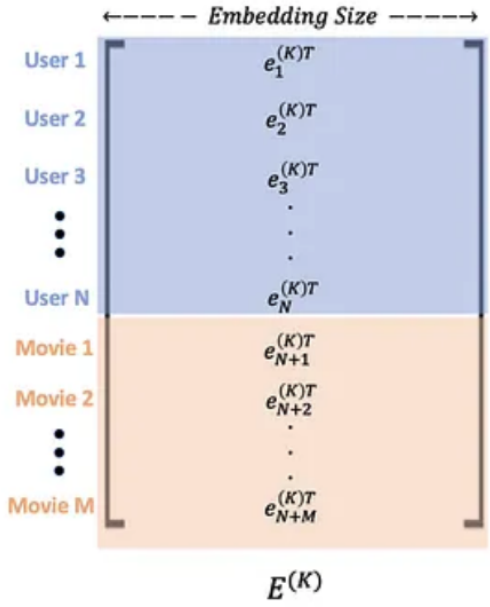


Figure 14: K-th layer embeddings, $E^{(K)}$

In collaborative filtering tasks, the lack of rich node features often hinders inductive GNNs from effectively transforming features into a latent space. However, LightGCN takes a transductive approach, directly learning embeddings for users and movies. This makes LightGCN particularly well-suited for the MovieLens dataset and other recommender systems that utilize simple user and item features.

2.3 Layer Combination and Model Prediction

In LightGCN, the only trainable parameters are the embeddings at the 0-th layer, denoted as $e_u^{(0)}$ for all users and $e_i^{(0)}$ for all items. Once these are established, embeddings for higher layers can be computed using the LGC defined in the above equation. After K layers of LGC, we combine the embeddings from each layer to create the final representation for a user (or item).

$$e_u = \sum_{k=0}^{K} a_k e_u^{(k)}; e_i = \sum_{k=0}^{K} a_k e_i^{(k)}$$

Here, $\alpha_k \geq 0$ represents the significance of the k-th layer embedding in forming the final embedding. This can be treated as a hyperparameter for manual tuning or as a model parameter (e.g., output from an attention network) for automatic optimization. We suggest that uniformly setting α_k to $1/(K+1)$ generally yields strong performance.

The model prediction is defined as the inner product of user and item final representations:

$$\hat{y}_{ui} = e_u^T e_i$$

which is used as the ranking score for recommendation generation. This inner product measures the similarity between the user and movie, therefore allowing us to understand how likely it is for the user to like the movie.

The trainable parameters of LightGCN are only the embeddings of the 0-th layer, i.e., $\Theta = \{E^{(0)}\}$; in other words, the model complexity is the same as the standard matrix factorization (MF).

When choosing surrogate losses, it's important that they are differentiable and closely aligned with the original training objective. Binary loss is a commonly used function in classification tasks, where the model must categorize inputs into one of two predefined classes. It is calculated as the sum of the losses for all positive and negative terms. However, binary loss tends to push the scores of all positive instances higher than those of negative ones, which can lead to unnecessary penalties for model predictions, even when the training metric is optimal. This happens because binary loss is non-personalized, treating all positive instances as if they should universally have higher scores than negative ones, without considering individual users. Therefore, it's essential to adapt binary loss to be more personalized.

We employ the Bayesian Personalized Ranking (BPR) loss, which is a pairwise loss that encourages the prediction of an observed entry to be higher than its unobserved counterparts:

$$L_{BPR} = -\sum_{u=1}^{M} \sum_{i \in N_u} \sum_{j \notin N_u} ln\sigma(\hat{y}_{ui} - \hat{y}_{uj}) + \lambda ||E^{(0)}||^2$$

where λ controls the L_2 regularization strength, y_{ui} represents the predicted score for a positive sample, while y_{uj} denotes the predicted score for a negative sample. The BPR loss is initially computed for a specific user u, and then we aggregate the losses across all users to obtain a system level BPR loss. We employ the Adam optimizer and use it in a mini-batch manner, $E^{(0)}$ is a matrix with column vectors being the 0-th layer embeddings to learn.

3 Implementation

The project is implemented with PyTorch-Geometric library and stored on Colab at link

For intralayer neighborhood aggregation, we implement a custom nn.MessagePassing layer (a generic GNN layer), LightGCNConv, takes the weighted sum of embedding embeddings.

```
def forward(self, x: Tensor, edge_index: Adj) -> Tensor:
        """Performs neighborhood aggregation for user/item embeddings."""
        # setup the matrix
        user_item = \
                torch.zeros(self.num_users, self.num_items, device=x.device)

        #
        # set value to 1 if there is a link
        user_item[edge_index[:, 0], edge_index[:, 1]] = 1

        # count number of user's neighbor
        user_neighbor_counts = torch.sum(user_item, axis=1)
        # count number of item's neighbor
        item_neightbor_counts = torch.sum(user_item, axis=0)

        # Compute weight for aggregation: 1 / sqrt(N_u * N_i)
        weights = user_item / torch.sqrt(
                user_neighbor_counts.repeat(self.num_items, 1).T \
                * item_neightbor_counts.repeat(self.num_users, 1))

        # normalize the data, turn nan to 0
        weights = torch.nan_to_num(weights, nan=0)

        # inner product
        out = torch.concat((weights.T @ x[:self.num_users],
                            weights @ x[self.num_users:]), 0)
        return out
```

For inter-layer combination, we implement a custom torch.nn.Module model, LightGCN, that stacks multiple LightGCN layers and computes the final user and item embeddings by taking a weighted sum of the embeddings at all layers.

```
    def forward(self, x: Tensor, edge_index: Adj, *args, **kwargs) -> Tensor:
        xs: List[Tensor] = []

        #
        edge_index = torch.nonzero(edge_index)
        for i in range(self.num_layers):
            x = self.convs[i](x, edge_index, *args, **kwargs)
            if self.device is not None:
                x = x.to(self.device)
            xs.append(x)

        # stack all result
        xs = torch.stack(xs)

        # calculate alpha, alpha is a vecto
        self.alpha = 1 / (1 + self.num_layers) * torch.ones(xs.shape)
        if self.device is not None:
            self.alpha = self.alpha.to(self.device)
            xs = xs.to(self.device)

        # get final sum
        x = (xs * self.alpha).sum(dim=0)  # Sum along K layers.
        return x
```

For each user, we randomly sample n positive and negative movie examples to include in the training, validation, or test set. The value of n is a parameter that can be specified and tuned.

To assess training progress and model performance, we calculate the top K ground truth items that the user likes and dislikes.

4 Insight

We know that precision measures the accuracy of positive predictions, while recall measures the completeness of positive predictions.In benchmarking recommendation systems, we typically avoid using overall accuracy because precision is more relevant when there are many relevant items but limited user attention. For example, when presenting only the top 5 movie recommendations from a pool of thousands, Precision at K effectively measures the quality of that shortlist.

- Precision at K is the ratio of correctly identified relevant items within the total recommended items inside the K-long list. Simply put, it shows how many recommended or retrieved items are genuinely relevant.
- Recall at K measures the proportion of correctly identified relevant items in the top K recommendations out of the total number of relevant items in the dataset. In simpler terms, it indicates how many of the relevant items we could successfully find. It measures the system's ability to retrieve all relevant items in the dataset.

We achieved a top 10 test precision of 0.0890 with 20 and 500 samples per user. However, plotting the training curves for both models, it becomes clear that training converges much faster with a larger number of samples per user. This is due to the random sampling process; we use only a small portion of over six thousand movies for training. A smaller sample size can lead to significant variance based on the selected subset, as some subsets may be more informative than others. Therefore, we generally recommend a larger sample size per user. However, setting it too high (e.g., 1000 samples per user) may not be practical, as the average number of movies rated by users is below 1000.

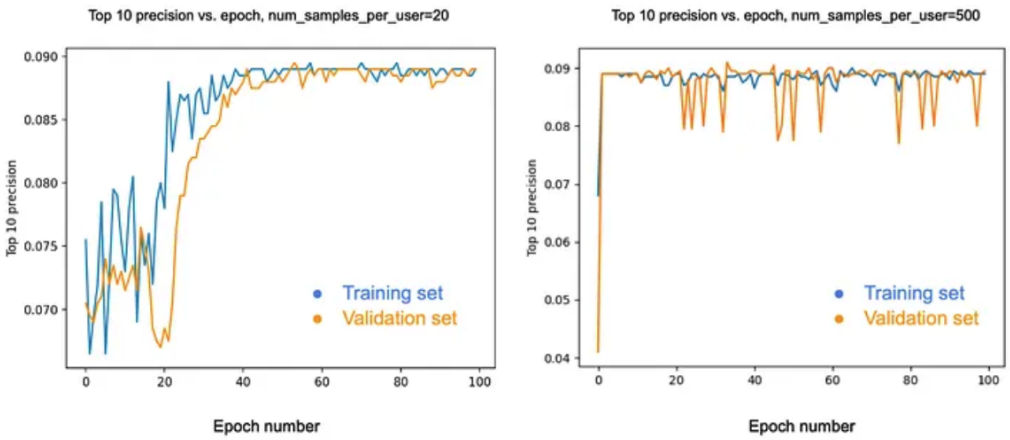
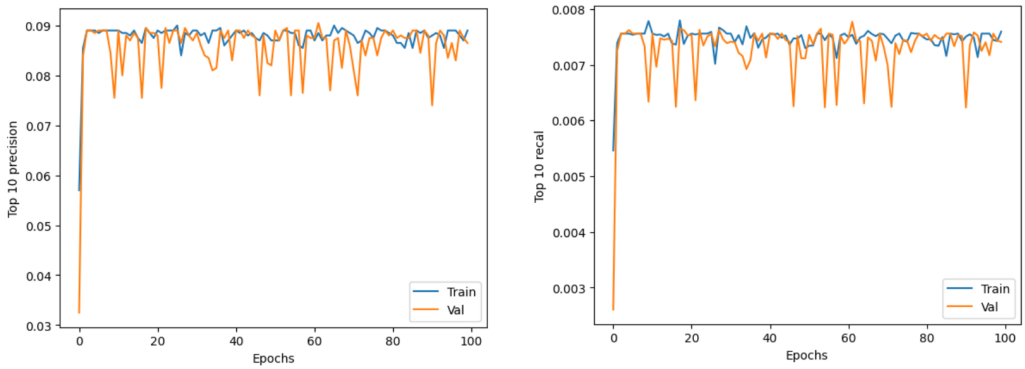


Figure 15: 20 samples per user vs 500 samples per user

We focus on the top 10 test precision when assessing model performance because it indicates how many of the recommended movies would likely receive positive ratings. With a top 10 precision score of 0.0895, this means that on average, if we recommend 10 movies, the user is expected to rate one of them 3 or higher.



(a) Top 10 precision

(b) Top 10 recal

We found that weight decay, which regulates the level of regularization, and the number of samples per user significantly impact our top-k recommendation precision. To further investigate, we performed ablation studies on each of these parameters individually.

weight_decay	Top 10 train precision	Top 10 train recall	Top 10 validation precision	Top 10 validation recall	Top 10 test precision	Top 10 test recall
0.0	0.0735	0.00572	0.0620	0.00488	0.0690	0.00535
0.001	0.0890	0.00756	0.0890	0.00756	0.0890	0.00756
0.01	0.0890	0.00756	0.0890	0.00756	0.0890	0.00756
0.1	0.0875	0.00769	0.0880	0.00752	0.0865	0.00736
1.0	0.0890	0.00756	0.0885	0.00754	**0.0895**	0.00754

Figure 17: Weight Decaying Effection

The trend in top-10 test precision with increasing num_samples_per_user is not very clear. We obtained a top-10 test precision of 0.0890 with both 20 samples per user and 500 samples per user.

num_samples_per_user	Top 10 train precision	Top 10 train recall	Top 10 validation precision	Top 10 validation recall	Top 10 test precision	Top 10 test recall
20	0.0885	0.00754	0.0885	0.00747	**0.0890**	0.00756
100	0.0865	0.00726	0.0870	0.00733	0.0850	0.00725
200	0.0790	0.00691	0.0770	0.00680	0.0770	0.00666
500	0.0890	0.00756	0.0895	0.00757	**0.0890**	0.00756

Figure 18: Amount of number of sample per user effection

References

[1] Gianluca Malato. (2021) Precision, recall, accuracy. How to choose?.

[2] EvidentlyAI. (2021) Precision and recall at K in ranking and recommendations.

[3] Jure Leskovec. (2024) CS224W: Machine Learning with Graphs..

[4] He, X., Deng, K., Wang, X., Li, Y., Zhang, Y., & Wang, M. (2020). LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation.

[5] PyTorch Geometric (PyG): https://pytorch-geometric.readthedocs.io/en/latest/

[6]F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. ACM Trans. Interact. Intell. Syst., 5(4), December 2015.

[7] Stanford CS224W GraphML Tutorials: link