LightGCN for Movie Recommendation

Coauthors: Feiyang Yu, LilyXLYang

Quinn Wang · Follow
Published in Stanford CS224W GraphML Tutorials · 11 min read · Feb 9, 2022

42 4

By Qinchen Wang, Xiaoli Yang, Feiyang (Kathy) Yu as part of the Stanford CS224W course project.

This blog post is based on the paper: "Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. Lightgcn: Simplifying and powering graph convolution network for recommendation, 2020."

What is movie recommendation?

If you have visited some online movie platforms, you may have noticed a section like "Guess You Love" that presents a list of interesting movies you have not seen before.

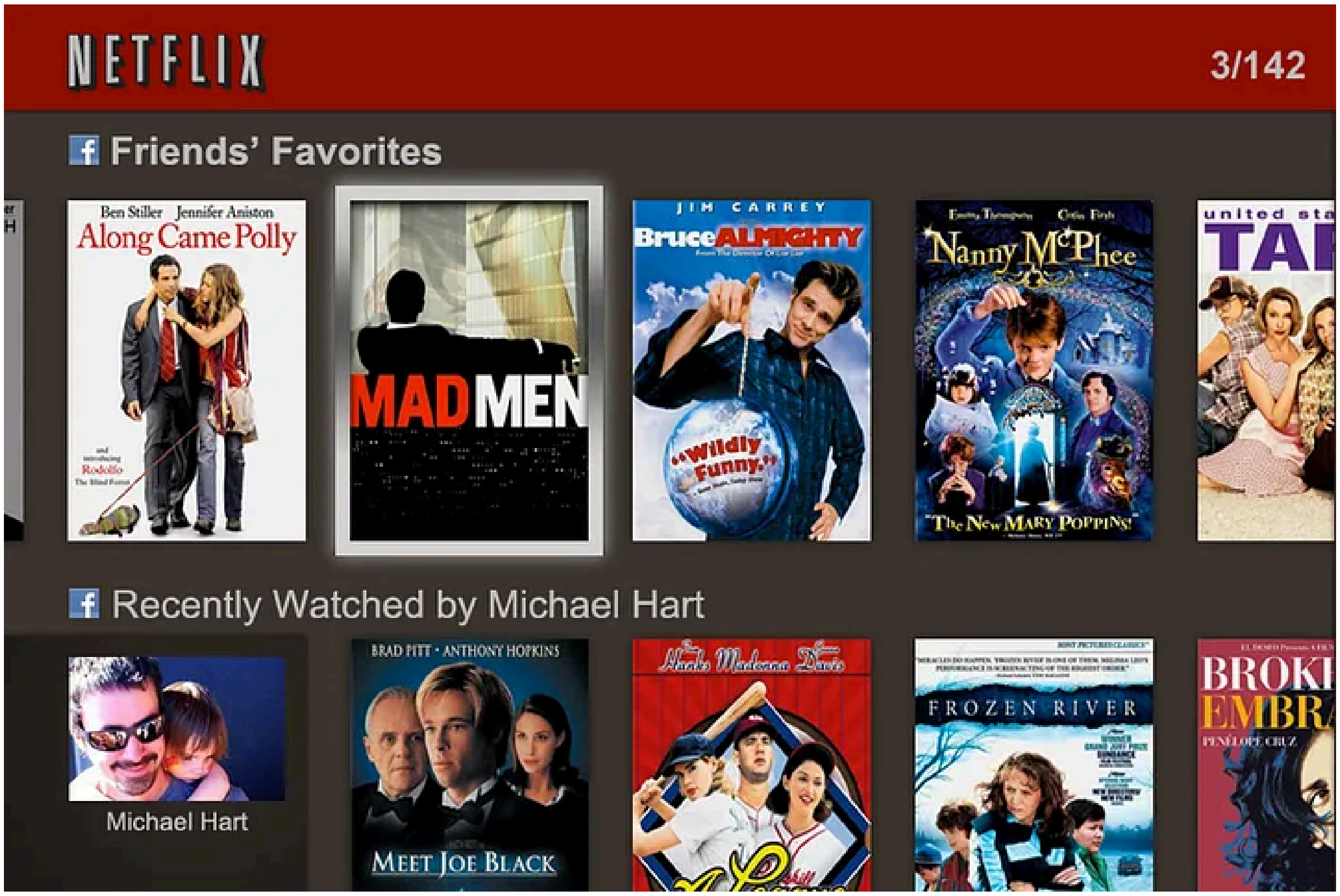


Image source by Griffin McElroy

This is called movie recommendation. Similar to this, you may often find shopping recommendations on Amazon and video recommendations on TikTok. These suggested lists all rely on the recommender system. Recommender system helps create a personalized list of item recommendations for users based on **1) how they interact with other users and 2) how they like or dislike certain items**. Employing user-user and user-item interactions in such a way to predict future recommendations is called **Collaborative Filtering (CF).**

CF is one of the most commonly explored approaches in the recommender system. It assumes that one user will like an item that another user with similar selecting behaviors has liked in the past and uses historical user-to-item interactions to predict potential interactions. Some early work in building recommender systems involves combining CF with Bayesian models [2], Decision Trees [3], Matrix Factorization (MF) [4,5], and clustering methods [6]. In particular, MF is one of the earlier attempts towards effective CF and remains popular today.

In this blog post, we will explore a state-of-the-art model fruited by the Graph Neural Network (GNN), LightGCN, and compare the performance of LightGCN and MF as movie recommender systems on the **MovieLens 1M Dataset**.

MovieLens Dataset and GNN Basics in Recommender Systems

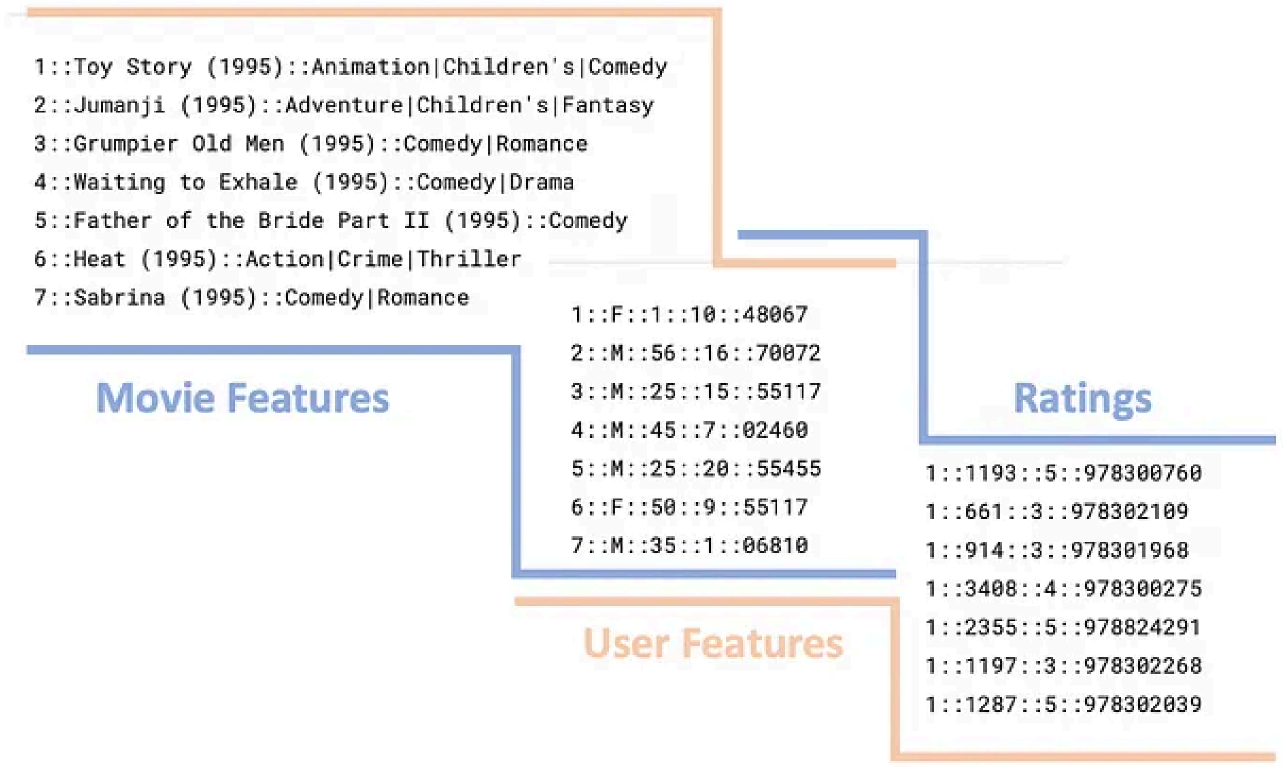
MovieLens

A great publicly available dataset for training movie recommenders is the MovieLens 1M [6] dataset. The MovieLens 1M dataset consists of 1 million **movie ratings of score 1 to 5**, from 6000 users and 4000 movies. There's also a family of MovieLens datasets, including sizes varying from 100K, 10M, 25M, etc., which allows you the flexibility to switch between various dataset sizes given your compute power and training goal.

MovieLens 1M is nice to train because: 1. it is a relatively well-established and clean dataset, which makes it easy to get started on training fast, 2. it is used as a benchmark for many publications; 3. features in this dataset are easily interpretable. Specifically, there are **user features and movie features**.

User features consist of gender, age, occupation and zip-code, whereas movie features consist of titles and genres. These features are commonly believed to be potentially correlated with people's inclinations.

With abundant movie ratings and a handful of user and movie features, we are ready to take a glimpse into how GNN recommender works and build a foundation for understanding LightGCN.



```
1::Toy Story (1995)::Animation|Children's|Comedy
2::Jumanji (1995)::Adventure|Children's|Fantasy
3::Grumpier Old Men (1995)::Comedy|Romance
4::Waiting to Exhale (1995)::Comedy|Drama
5::Father of the Bride Part II (1995)::Comedy
6::Heat (1995)::Action|Crime|Thriller
7::Sabrina (1995)::Comedy|Romance
```

```
1::F::1::10::48067
2::M::56::16::70072
3::M::25::15::55117
4::M::45::7::02460
5::M::25::20::55455
6::F::50::9::55117
7::M::35::1::06810
```

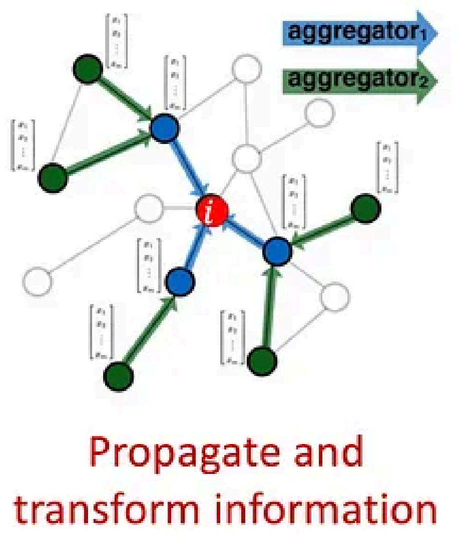
```
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
1::1197::3::978302268
1::1287::5::978302039
```

A Glimpse into GNN

GNN is a general name for a set of models that considers the problem setup from a graph perspective and utilizes neural networks to make predictions.

In GNN, entities are usually treated as nodes, and relationships between entities are described by edges. One could further supplement the graph with additional information by attaching node features and edge features to the graph. GNN generates embeddings while taking into account graph structures. Specifically, GNN utilizes the embeddings of a central node's neighbors to update the central node's embedding; neighbors can be defined as being some distance, or say "hop," away from the central node. This allows the GNN to create similar embeddings for nodes that are more closely related from both a local and a relatively more global perspective.

During the training process, a node sends "messages" as vector embeddings to its neighboring nodes, and receives messages from all its neighbors. For each node, messages are aggregated through a permutation equivariant function, which means that the output of such a function will be the same regardless of its input ordering. These functions include summation, mean, max, etc. This process is called **neighborhood aggregation**. The output from the aggregation function can be transformed and used to update the node embedding, and the updated node embedding can be used to propagate information to the next layer of the GNN training process. The intuition behind such a training process is that, by passing messages through multiple hops in the neighborhood, each node learns an embedding that captures its relationship to its neighbors.

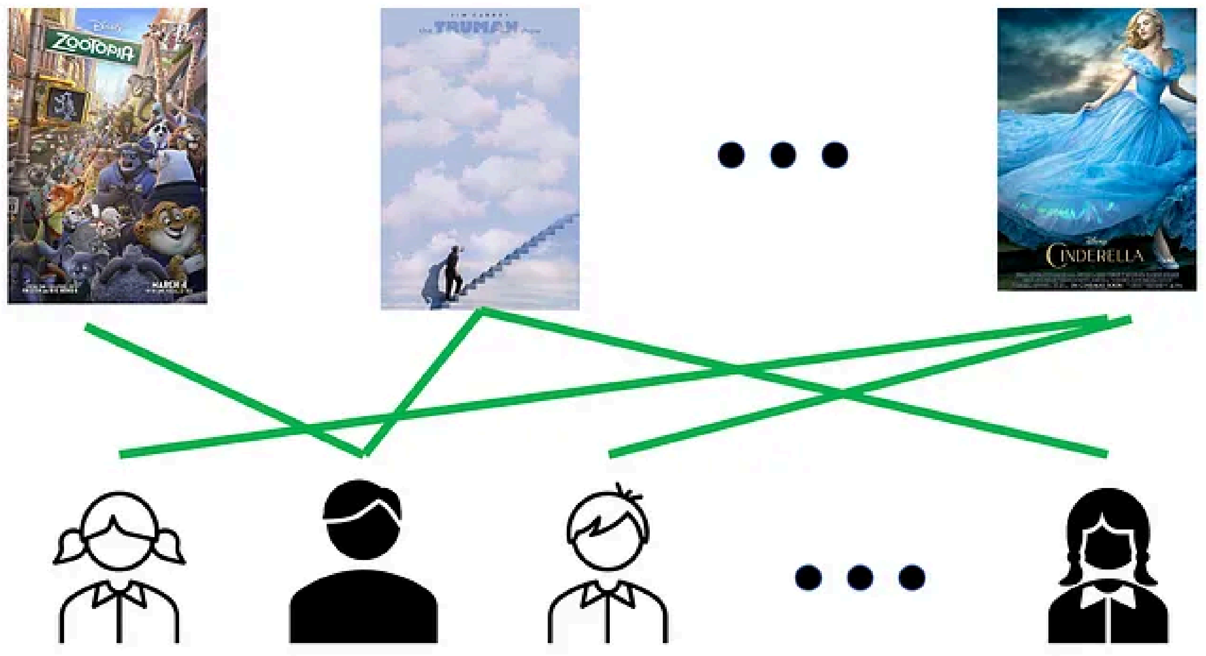


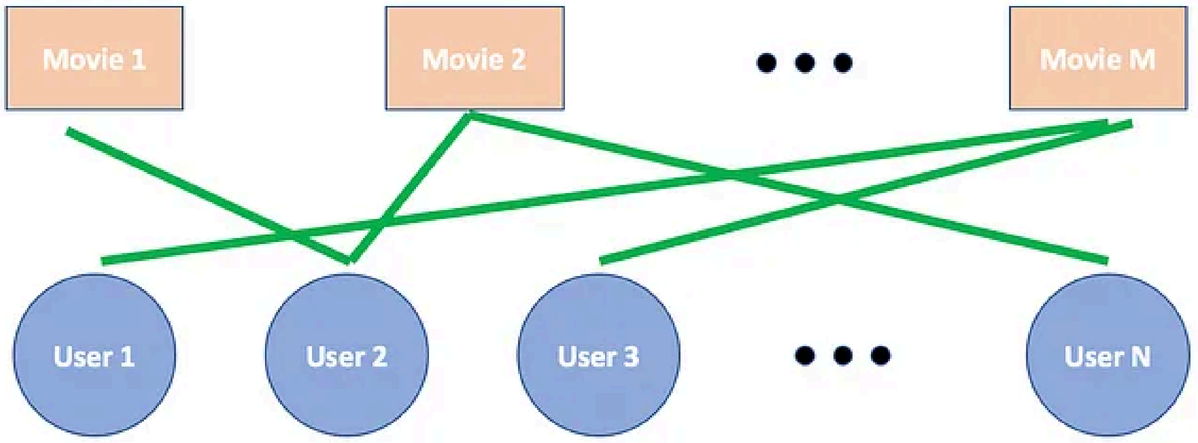
From Stanford CS224W: Machine Learning with Graphs

Among all instances of GNN, LightGCN is one that delivers state-of-the-art empirical performance on benchmarks for recommendations, including Gowalla, Yelp2018 and Amazon-Book. In the following section, we will introduce how LightGCN works with the MovieLens 1M dataset.

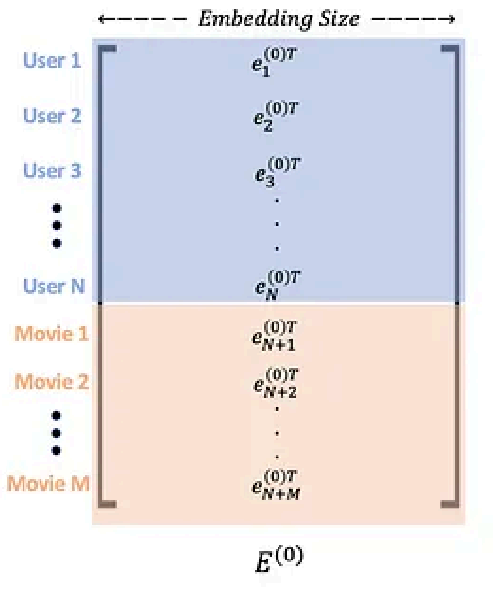
LightGCN Setup on MovieLens

Similar to the general GNN, when applying LightGCN to the MovieLens dataset, we treat the user-movie relationships as a **bipartite graph**. Specifically, we imagine users and movies as two types of nodes put at opposite sides with edges in between indicating user-movie interactions. In our case, we would like to predict whether a user likes a movie, so one way to summarize the data using a bipartite graph is: an edge exists if a user rates a movie higher than or equal to 3 and no edge exists if a user rates it lower than 3 or has never rated the movie.





As the last step in preparation, we randomly initialize an embedding matrix for users and movies, $E^{(0)}$, with shape [# of users + # of movies, 64], where 64 is the embedding size. **Note that parameters in this matrix are the only trainable parameters in LightGCN.**



Typically, in collaborative filtering tasks, non-rich node features make it difficult for inductive GNNs to learn feature transformations into a latent space. However, LightGCN is a transductive approach that directly learns embeddings for our users and movies. This makes LightGCN especially suitable for the MovieLens dataset, as well as any recommender system with simple user and item features.

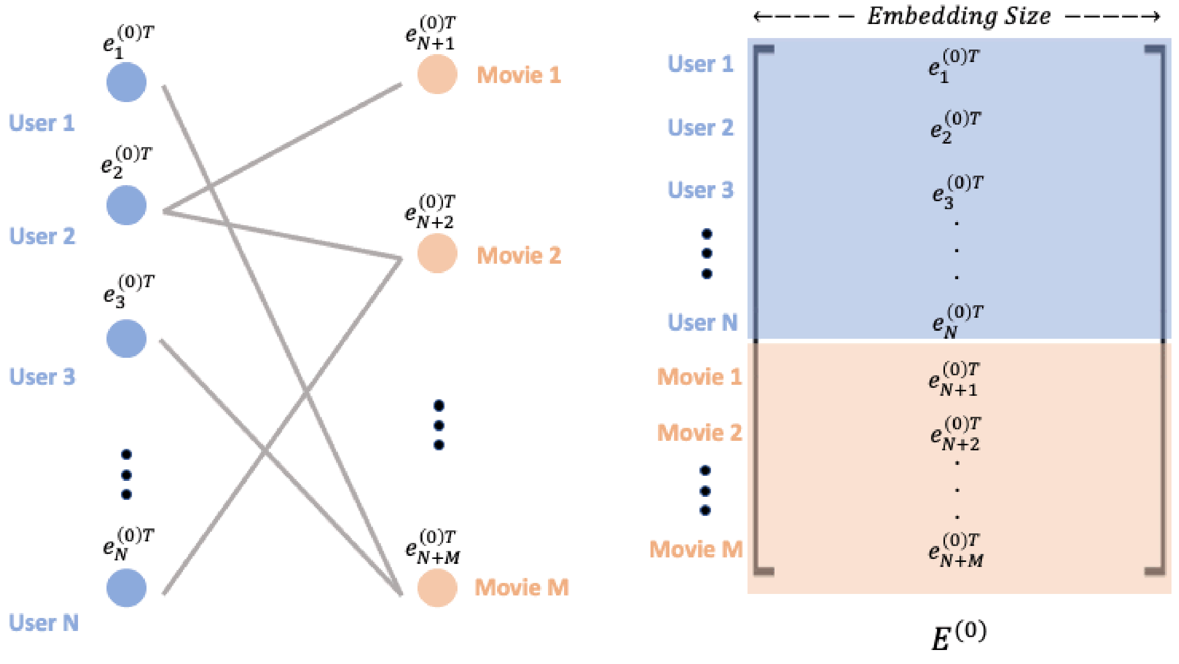
Journey into LightGCN Training

Now we are going to speed up. Fasten your seat belt!

LightGCN's secret lies in two key designs: **(1) intra-layer neighborhood aggregation; (2) inter-layer combination**. These concepts may seem intimidating at the first glance. Don't panic! Let's look at them step-by-step.

(1) Within each layer, for each user in the graph, we compute its updated embedding as the weighted sum of embeddings from all its neighboring items (movies).

See the animation below for exactly how that weighted sum is computed:



The exact equation for aggregators is the following:

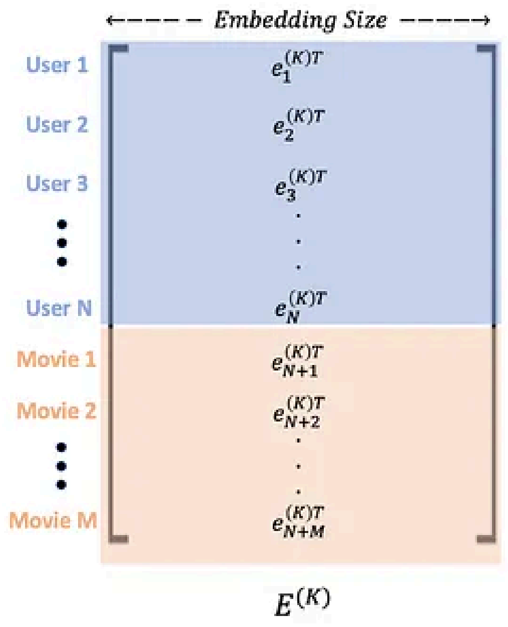
$$\mathbf{e}_u^{(k+1)} = \sum_{i \in N_u} \frac{1}{\sqrt{|N_u|}\sqrt{|N_i|}} \mathbf{e}_i^{(k)}$$

where $\mathbf{e}_u^{(k)}$ and $\mathbf{e}_i^{(k)}$ 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.

Similarly, for each item, the updated embedding is computed using a weighted sum of its neighboring users:

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

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



(2) At layer combination, instead of taking the embedding of the final layer, LightGCN computes **a weighted sum of the embeddings at different layers:**

The equations for this weighted summation are as follows:

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

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

with $\alpha \geq 0$. Here, alpha values can either be learned as network parameters, or set as empirical hyperparameters. It has been found that $\alpha = 1/(K + 1)$ works well.

Finally, LightGCN predicts based on the inner product of the final user and item (movie) embeddings:

$$\hat{y}_{ui} = \mathbf{e}_u^T \mathbf{e}_i$$

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.

To train the LightGCN model, we need an objective function that aligns with our goal for movie recommendation. We use the **Bayesian Personalized Ranking (BPR) loss**, which encourages observed user-item predictions to have increasingly higher values than unobserved ones, along with L_2 regularization:

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

where $\mathbf{E}^{(0)}$ is a matrix with column vectors being the 0-th layer embeddings to learn.

How do we implement the model using PyTorch?

Now that you understand how lightGCN works, let's implement it using PyTorch and see it in action. You may find all code and a walkthrough of the implementation in *[this Colab notebook]*.

Specifically, we use the **PyTorch-Geometric** library, which is a GNN dedicated library built on top of PyTorch that allows you to flexibly implement GNN applications. It works well with **GraphGym**, a platform that offers modularized GNN pipelines and experiment management features.

(1) For intra-layer neighborhood aggregation, we implement a custom torch_geometric.nn.MessagePassing layer (a generic GNN layer), **LightGCNConv**, that takes the weighted sum of embedding embeddings. A code snippet of LightGCNConv is shown below:

```python
def forward(self, x: Tensor, edge_index: Adj) -> Tensor:
    """Performs neighborhood aggregation for user/item embeddings."""
    user_item = \
            torch.zeros(self.num_users, self.num_items, device=x.device)
    user_item[edge_index[:, 0], edge_index[:, 1]] = 1
    user_neighbor_counts = torch.sum(user_item, axis=1)
    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))
    weights = torch.nan_to_num(weights, nan=0)
    out = torch.concat((weights.T @ x[:self.num_users],
                        weights @ x[self.num_users:]), 0)
    return out
```

lightgcn_conv.py hosted with ❤ by **GitHub** **view raw**

(2) 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. A code snippet for taking said sum is shown below:

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

    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)
    xs = torch.stack(xs)

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

lightgcn.py hosted with ❤ by **GitHub** **view raw**

For each user, we randomly sample *n* positive-negative movie examples and add them to the training, validation or test set. *n* is a parameter that we can specify and tune.

To evaluate training progress and model performance, we compute the **top K**

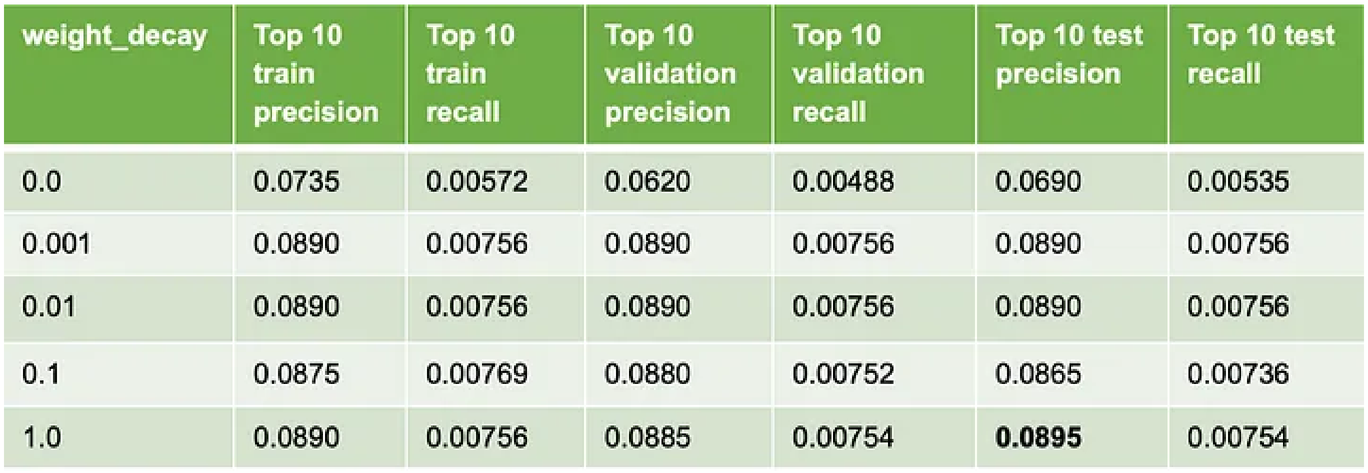
ground truth items that the user likes and dislikes.

How did we do? (Results)

We observed that the parameters **weight decay** (which controls the degree of regularization) and **number of samples per user** have the most effect on our top k recommendation precision. We conducted ablation studies on these two parameters separately.

# 1

We ran an experiment with different values of weight decay, while keeping other parameters as: *{'num_samples_per_user': 20, 'num_users': 200, 'epochs': 100, 'batch_size': 128, 'lr': 0.001, 'embedding_size': 64, 'num_layers': 5, 'threshold_pred': 0.9, 'minibatch_per_print': 100, 'epochs_per_print': 1, 'val_frac': 0.2, 'test_frac': 0.1, 'K': 10}*, and we trained 5 different models for **weight_decay** [0.0, 0.001, 0.01, 0.1, 1.0]. The results are shown as follows:



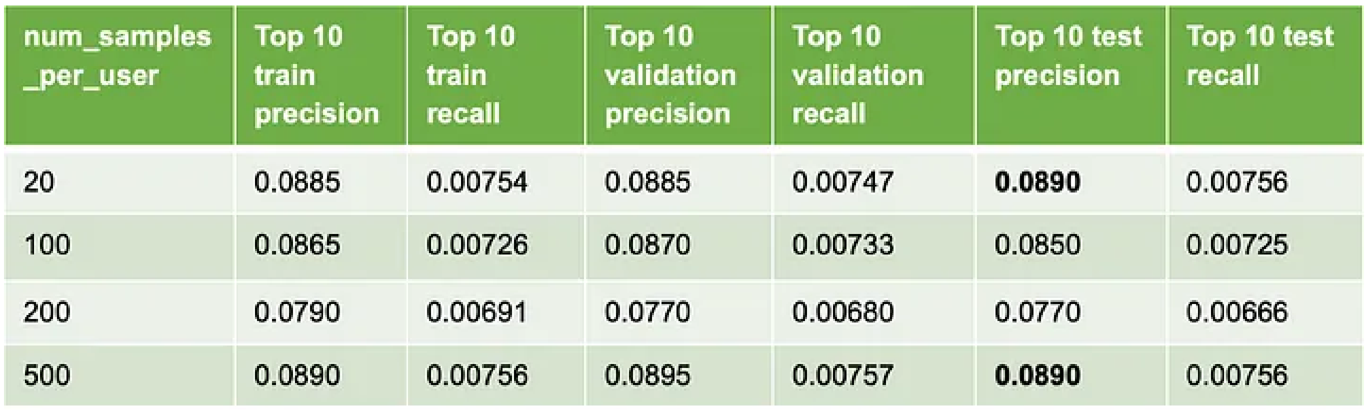
weight_decay	Top 10 train precision	Top 10 train recall	Top 10 validation precision	Top 10 validation recall	Top 10 test precision	Top 10 test recall
0.0	0.0735	0.00572	0.0620	0.00488	0.0690	0.00535
0.001	0.0890	0.00756	0.0890	0.00756	0.0890	0.00756
0.01	0.0890	0.00756	0.0890	0.00756	0.0890	0.00756
0.1	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

Results table with different weight_decay

We are interested in the top 10 test precision when evaluating model performance, because top 10 precision tells us "if our model were to recommend 10 movies to our user, what's the average number of movies our user would like, out of those top 10?" With a weight_decay of 1.0, we achieve the highest top 10 test precision score of 0.0895, which means that on average, out of the 10 movies we recommend, our user will give one good rating of 3 or above!

2

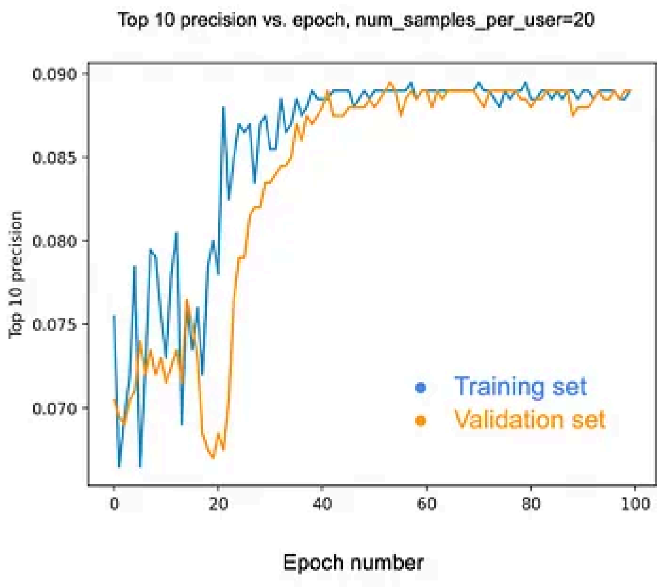
We also ran an experiment with different number of samples per user, while keeping the other parameters as *{'num_users': 200, 'epochs': 100, 'batch_size': 128, 'lr': 0.001, 'weight_decay': 0.1, 'embedding_size': 64, 'num_layers': 5, 'threshold_pred': 0.9, 'minibatch_per_print': 100, 'epochs_per_print': 1, 'val_frac': 0.2, 'test_frac': 0.1, 'K': 10}*, and we trained 4 different models for **num_samples_per_user** [20, 100, 200, 500]. The results are shown as follows:

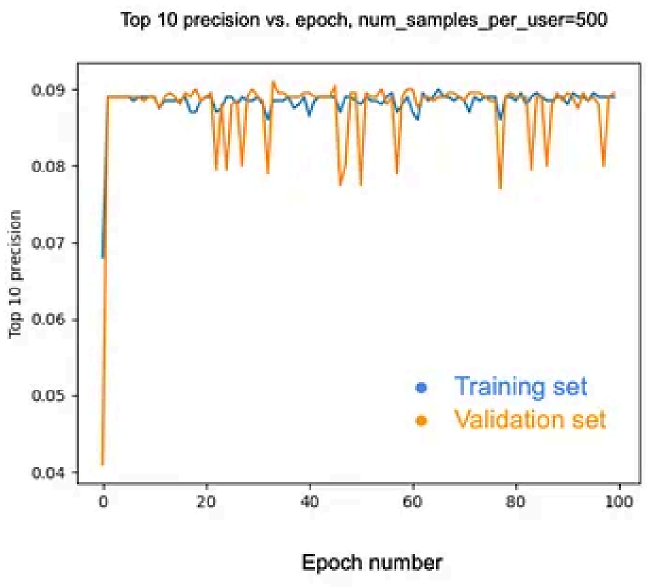


num_samples_per_user	Top 10 train precision	Top 10 train recall	Top 10 validation precision	Top 10 validation recall	Top 10 test precision	Top 10 test recall
20	0.0885	0.00754	0.0885	0.00747	**0.0890**	0.00756
100	0.0865	0.00726	0.0870	0.00733	0.0850	0.00725
200	0.0790	0.00691	0.0770	0.00680	0.0770	0.00666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

Results table with different num_samples_per_user

There isn't as much of a clear trend in terms of top 10 test precision increase under the effect of increasing num_samples_per_user. We achieved a top 10 test precision of 0.0890 both with 20 samples per user, and 500 samples per user. However, if we plot the training curve for both models:





we can see that when the number of samples per user is larger, training converges much faster. This is because the sampling process is random — out of over six thousand movies, we only sample a small portion to use in our training. If this sampling size is small, the amount of variance depending on which subset of data we sampled will be large — some subset might be more informative than others. So we generally recommend setting a relatively large sample size per user. However, going too large (for example 1000 samples per user) might not be reasonable for this dataset, because the average number of movies the users rated is below 1000.

Finally, we want to compare our results using LightGCN with the classic Matrix Factorization technique. We used the PARAFAC implementation with TensorLy. The only parameter we had to tune for MF is the matrix rank, and we found rank=8 is a good value for it to achieve good results compared with higher or lower rank values. The top 10 precision we got with MF on the test set is 0.0530, and the top 10 recall is 0.00542, which are comparably lower than what we achieved with LightGCN.

Congratulations! You've now implemented your first GNN (Conclusion)

In this blog post, we learned about some of the basic intuitions behind GNN, and delved into the technical details of LightGCN. We have also applied LightGCN to a real-life task that affects millions of people's recreation time — movie recommendation. The power of GNN extends beyond recommendation tasks, and we hope this is a gentle introduction that gets you excited about the potential of this powerful network structure!

Want to try more? Here are some next-steps (Resources)

- "LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation": https://arxiv.org/abs/2002.02126
- LightGCN author code: https://github.com/gusye1234/LightGCN-PyTorch
- Stanford CS224W: http://web.stanford.edu/class/cs224w/
- PyTorch Geometric (PyG): https://pytorch-geometric.readthedocs.io/en/latest/
- GraphGym: https://github.com/snap-stanford/GraphGym
- DeepSNAP: https://snap.stanford.edu/deepsnap/
- MovieLens: https://grouplens.org/datasets/movielens/
- TensorLy matrix factorization: http://tensorly.org/dev/modules/generated/tensorly.decomposition.parafac.html#tensorly.decomposition.parafac
- "PARAFAC. Tutorial and applications": https://www.sciencedirect.com/science/article/abs/pii/S016974399700032

4

References

[1] Koji Miyahara and Michael J. Pazzani. Collaborative filtering with the simple bayesian classifier. In PRICAI, 2000

[2] Amir Gershman, Amnon Meisels, Karl-Heinz Lüke, Lior Rokach, Alon Schclar, and Arnon Sturm. A decision tree based recommender system. In Gerald Eichler, Peter Kropf, Ulrike Lechner, Phayung Meesad, and Herwig Unger, editors, 10th International Conferenceon Innovative Internet Community Systems (I2CS) — Jubilee Edition 2010 –, pages 170–179, Bonn, 2010. Gesellschaft für Informatik e.V.

[3] Robin Devooght, Nicolas Kourtellis, and Amin Mantrach. Dynamic matrix factorization with priors on unknown values, 2015.

[4] Xin Guan, Chang-Tsun Li, and Yu Guan. Matrix factorization with rating completion: An enhanced svd model for collaborative filtering recommender systems. IEEE Access, 5:27668–27678, 2017.

[5] Hafed Zarzour, Ziad Al-Sharif, Mahmoud Al-Ayyoub, and Yaser Jararweh. A new collaborative filtering recommendation algorithm based on dimensionality reduction and clustering techniques. In 2018 9th International Conference on Information and Communication Systems (ICICS), pages 102–106, 2018.

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

Graph Neural Networks Recommendation System Pytorch Geometric Cs224w

Written by Quinn Wang

54 Followers · Writer for Stanford CS224W GraphML Tutorials

Follow

Data analyst with an interest in machine learning. Passionate about understanding the theoretical backings of ML algorithms.