PMLDL Assignment 2 Report

Introduction

Our task is to create a recommender system on MovieLens 100K dataset.

The dataset contains:

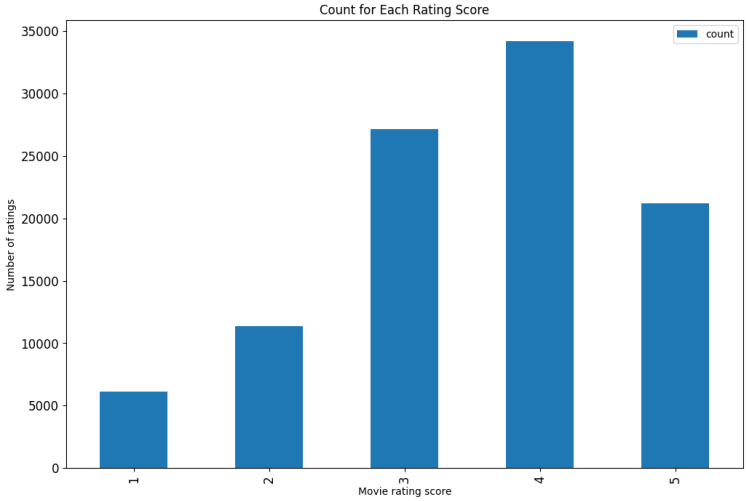
- 100,000 ratings on the scale of 1 to 5 from 943 users on 1682 movies
- information about items (movies): movie id, movie title, release date, video release date, IMDB URL, and genres
- information about the users: user id, age, gender, occupation, zip code.

It was decided to use and modify the code from the Lab 11 as it solves exactly the same problem. The code uses only rating data from the dataset and more specifically only positive ratings (3 and higher). It uses the data to train two different GNN (Graph Neural Network) models.

Data analysis

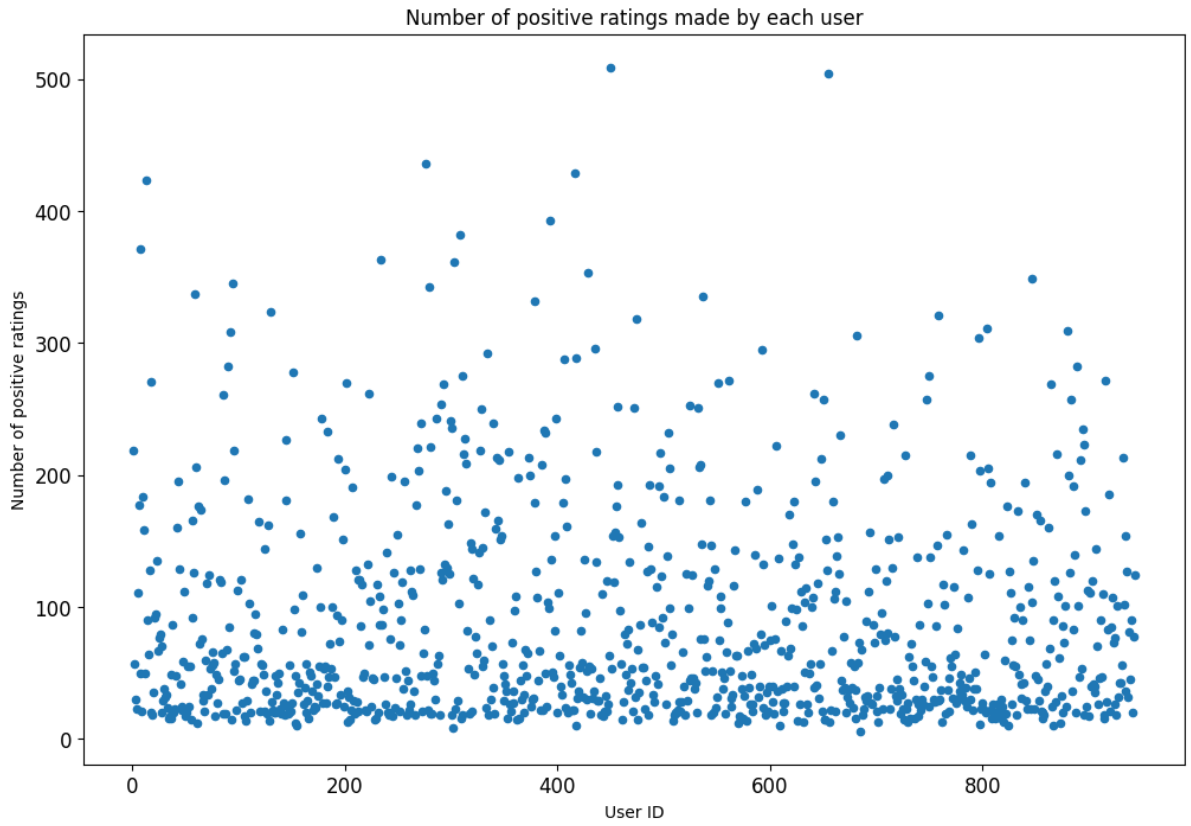
The code from the Lab 11 uses the relevant/irrelevant concept by considering items with rating 3 or more as relevant.

First, it is wise to check the distribution of ratings, in order to determine the length of the filtered dataset.



As you can see, if we leave only items with rating 3 or higher, we actually preserve most of the dataset.

Second, it would be good to ensure that all users have films that they like.



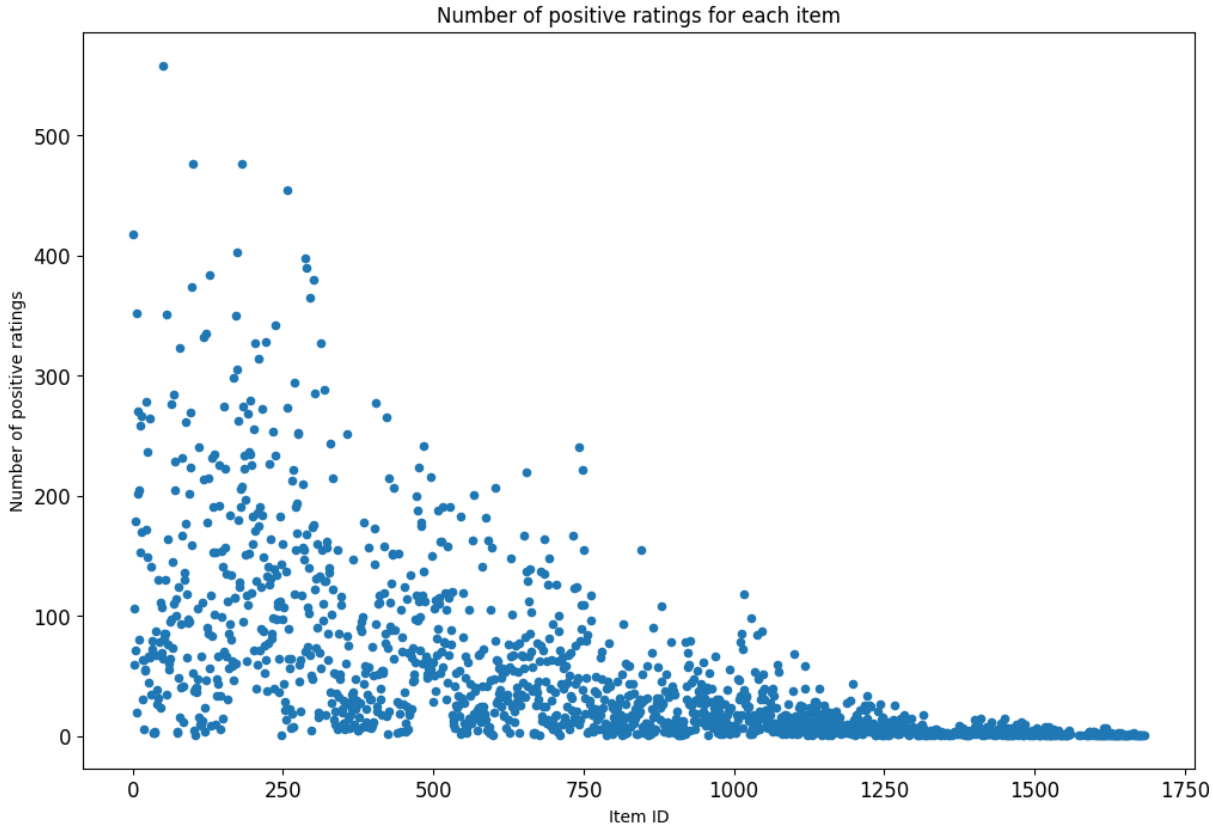
Now, as we are doing train/test split, it would be good to check if all users and items are present in the train dataset.

```
[ ]  train_user_ids = train_df['user_id'].unique()
     train_item_ids = train_df['item_id'].unique()

     n_users = len(train_user_ids)
     n_items = len(train_item_ids)

     print("Unique users:", n_users)
     print("Unique items:", n_items)

     Unique users: 943
     Unique items: 1546
```

```
[ ]  print(train_user_ids.min(), train_user_ids.max())
     print(train_item_ids.min(), train_item_ids.max())

     1 943
     1 1682
```

As you can see not all items are present.

Now, as we know how many positive ratings all users have made, it would be also good to know how many likes all items have received:



The situation is different here. As you can see, a lot of items have received small number of positive ratings. In more details:

```
[126] print(grouped['count'].min(), grouped['count'].max())

      1 558
```

```
[128] print((grouped['count'] == 1).sum())

      135
```

135 items have only one like. And with probability of 20% we will not see them in the training set.

Whereas, for the previous graph detailed statistics look like this:

```
[183] print(grouped['count'].min(), grouped['count'].max())

      6 509
```

Which means every user made at least 6 positive ratings. Therefore there are 6 entries in the dataset and there is a high chance this user will be present in the training set after train/set split.

So we do relabeling to make the highest item label to be equal to the number of items.

Model implementation

In the Lab 11 code two GNN models are used:

1. Stack of NGCF convolutional layers with Xavier initialization.
2. Stack of LightGCN convolutional layers with normal initialization.

Both models employ a message passing mechanism for embedding propagation. The only difference is that LightGCN removes the learnable linear layers, non-linear activation, and dropout.

Model advantages and disadvantages

The main advantage is convenience of using GNN for the recommender system task. The forward method is very simple and easy to understand. It requires minimal data preprocessing. The data is in format of user-item interactions which are conveniently represented as graph edges.

The second advantage is the speed of training. After 5 minutes of training model already shows overfitting indicating sufficient training.

However, there are disadvantages. Incorporating user data (such as age and gender) is not straightforward. The second disadvantage is the inability to add new users. As we don't have embeddings for new users, it is not possible to make prediction without retraining the model.

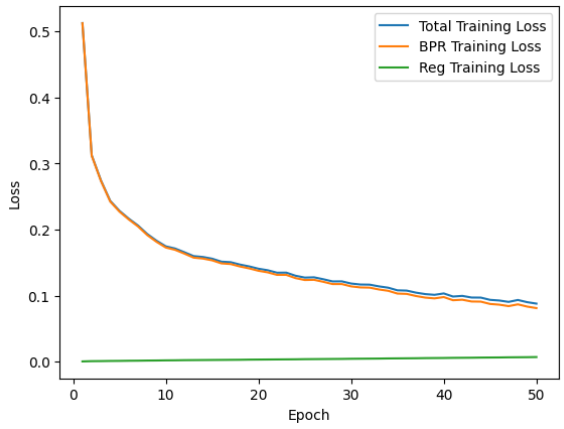
Training process

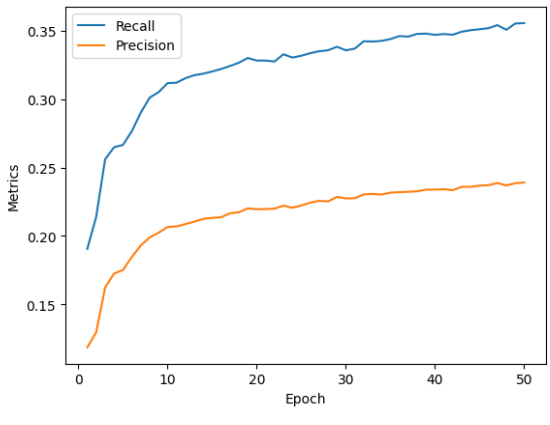
BPR (Bayesian Personalized Ranking) loss is used for training together with Regularization loss (to penalize the model for large weights to prevent overfitting).

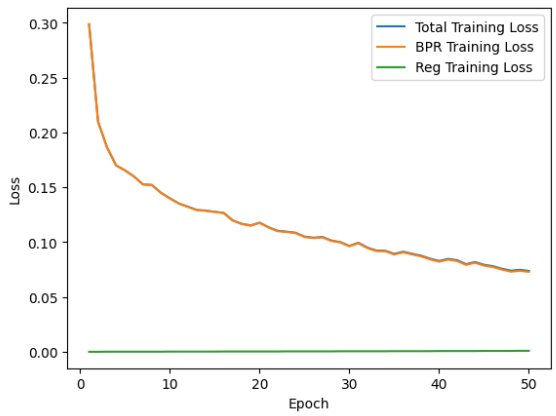
The two losses are stored separately for visualizations, but the backpropagation uses the sum of the two losses.

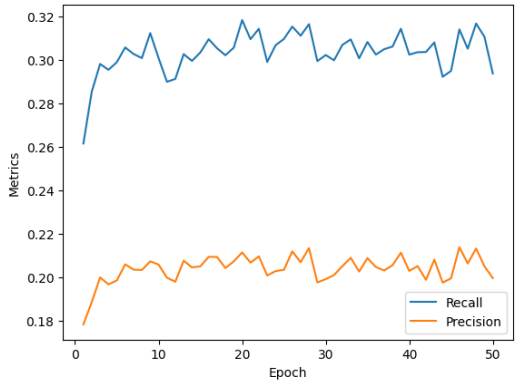
Each training epoch is followed by evaluation which stores outputs from the **Evaluation function.**

3 layers (top: model with LightGCN layers, bottom: model with NGCF layers):

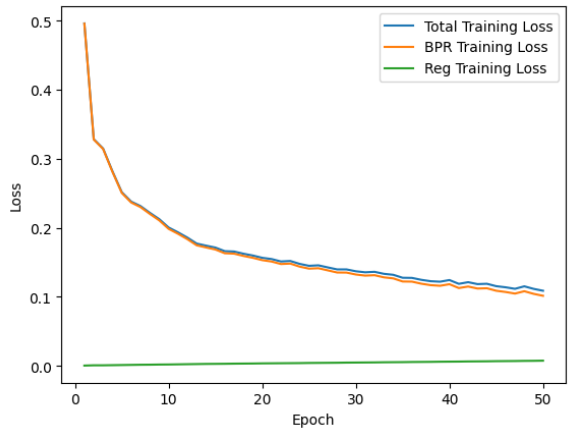


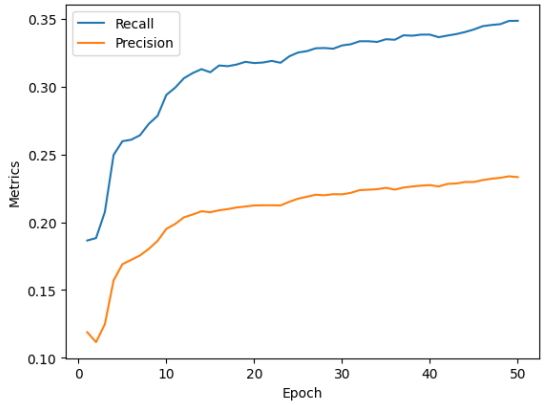


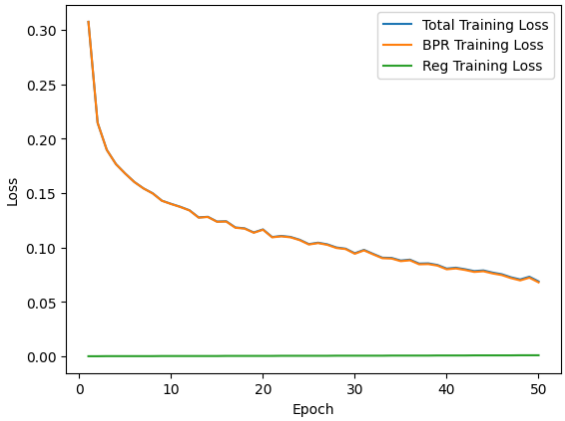


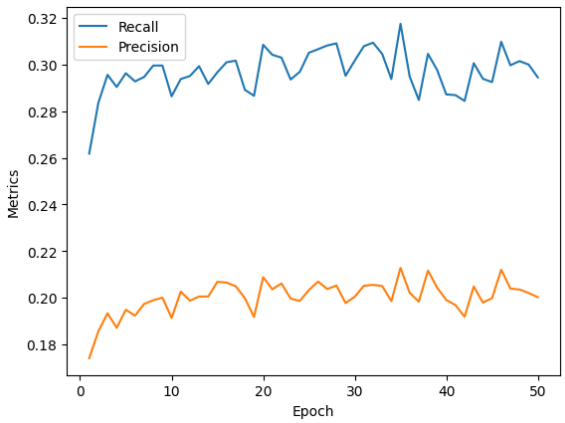


4 layers (top: model with LightGCN layers, bottom: model with NGCF layers):

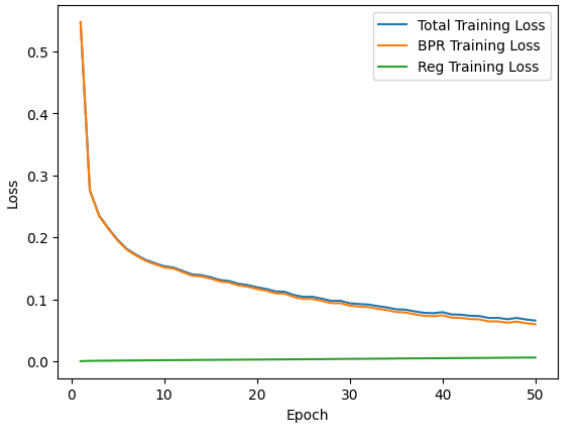


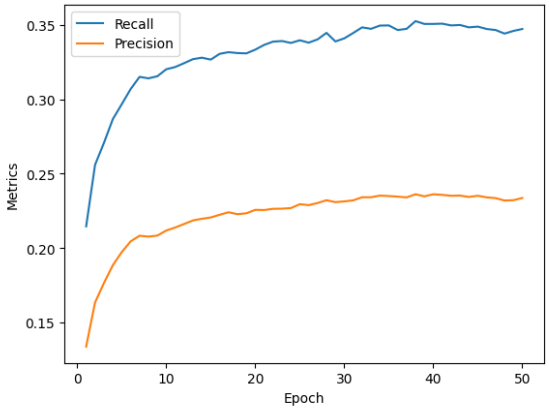


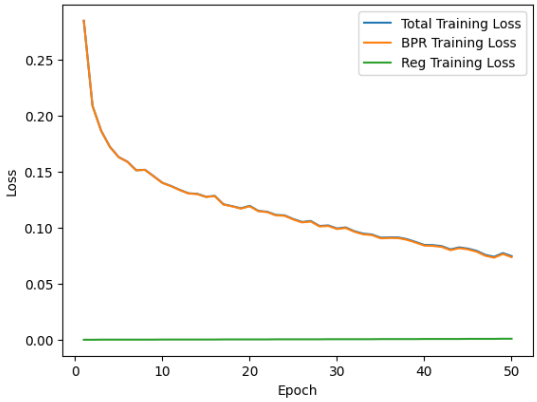


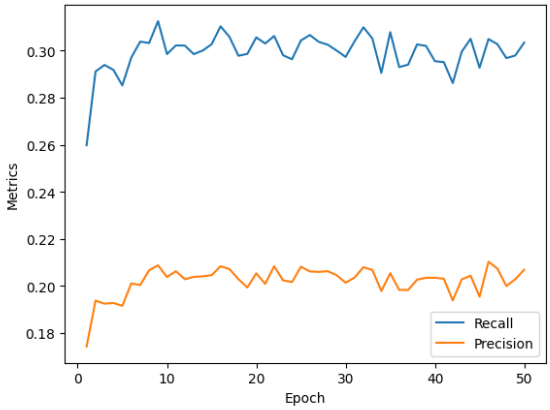


2 layers (top: model with LightGCN layers, bottom: model with NGCF layers):

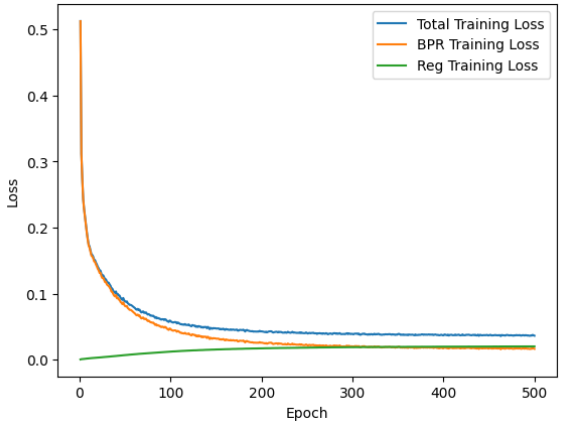


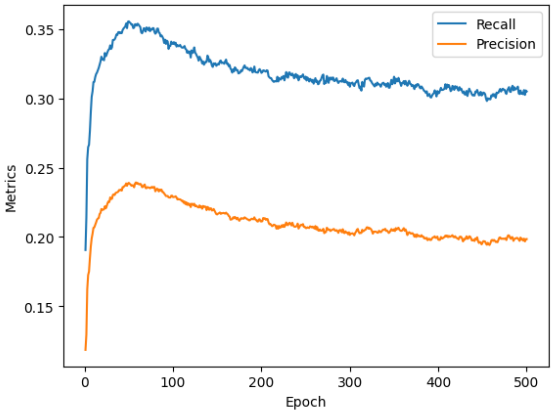


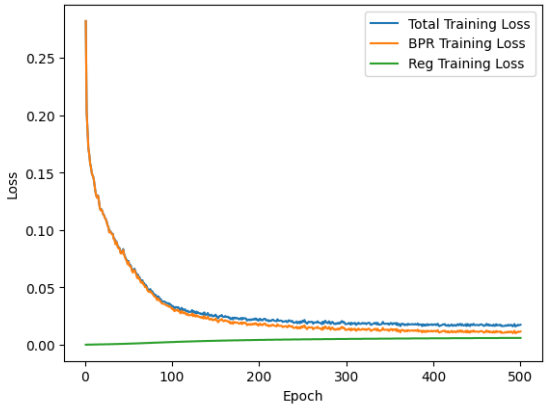


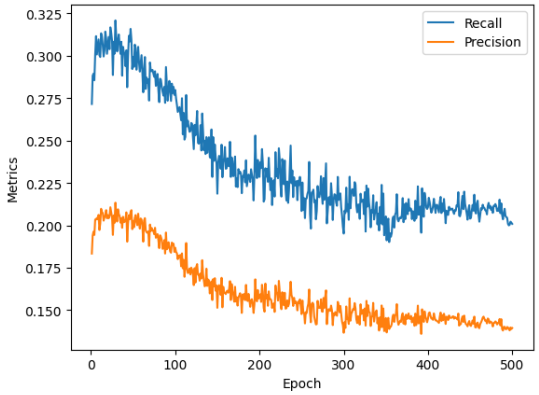


3 layers and 500 epochs (top: LightGCN layers, bottom: NGCF layers):



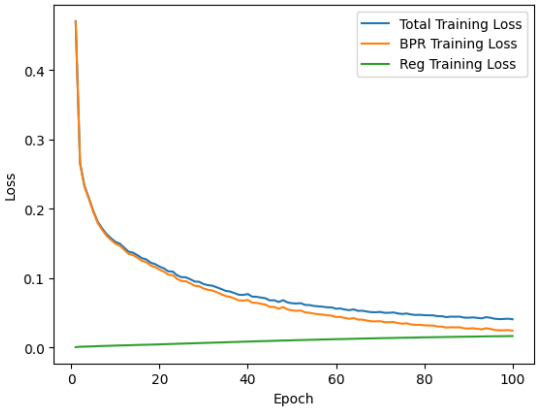


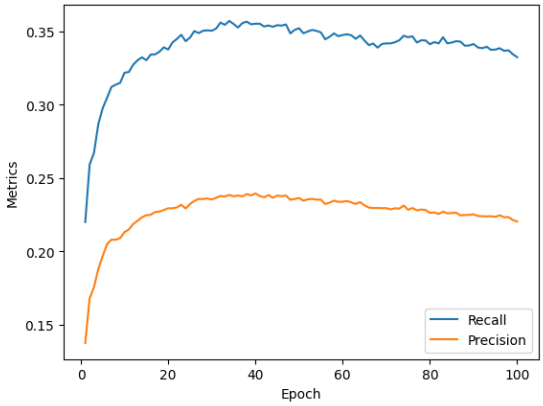


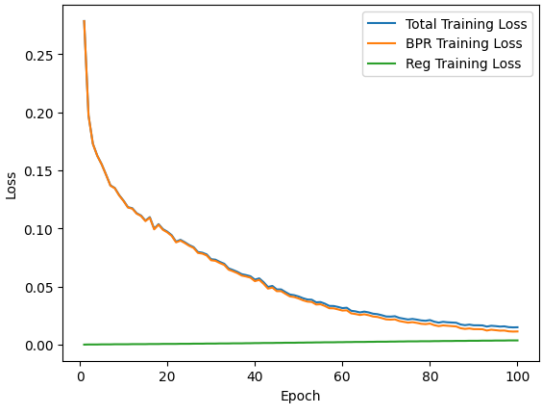


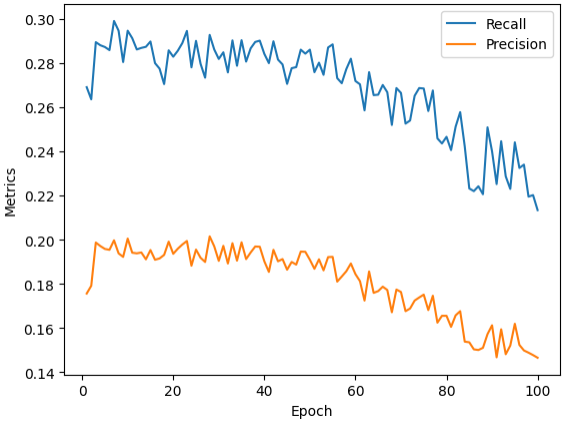
Looks like 50 epochs was exactly right number.

3 layers, embedding dimension 128 (instead of 64), and 100 epochs (top: model with LightGCN layers, bottom: model with NGCF layers):









Now metrics are started to show decrease even during 50 epochs. Which means the initial values of hyperparameters were in fact the optimal values.

Evaluation function

`get_metrics(user_Embed_wts, item_Embed_wts, n_users, n_items, train_data, test_data, K)`

Input:

`user_Embed_wts`, `item_Embed_wts` are result of split of output of model's `forward` method call

`n_users`: number of unique users

`n_items`: number of unique items

`train_data`, `test_data`: datasets in form of Pandas DataFrame

`K`: number of the most relevant items used in calculations

Output:

- Mean (among all users) recall at K
- Mean (among all users) precision at K

Internal implementation:

1. Multiplication of user embeddings and item embedding to get `relevance_score` matrix. Resulting shape: `(n_users, n_items)`.
2. Setting all entries in `relevance_score` that are present in training data to zero. Because we don't want entries from the training set to be present among the top K.
3. Computing top K items from `relevance_score` and storing them to a DataFrame (with columns: `'user_ID','top_rlvnt_itm'`) and also concatenating column with user's liked items from the test set.
4. Calculating for every row recall at K, precision at K using top K items found and items that user actually liked.
5. Returning average of those.

Results

The best result of the training you can see on the picture above. Both precision and recall are calculated using `K = 20`.

```
max(light_precision), max(light_recall)
```

```
(0.239, 0.3556)
```

```
max(ngcf_precision), max(ngcf_recall)
```

```
(0.2114, 0.3195)
```

As you can see, LightGCN showed better results in both precision and recall even though the model is less complicated. The 24% precision indicates that every fourth movie out of 20 that we recommended user would like to watch (on average). And the 35% recall means in these 20 movies that we recommended, we have 35% of all movies user would like to watch.

Conclusion: two GNN models were tested for recommender system task with different sets of hyperparameters, and given simplicity of both models, the result (measured in mean recall at K and mean precision at K) is considered decent.