Technical Report – Graph Neural Networks driven Recommender Systems

Vertiefungsprojekt I (HS22), Master of Eng. in Data Science

Student: Roman Loop

Academic Supervisor: Shao Jü Woo

Project duration: 10. October 2022 to 31. January 2023

Introduction

Recommender systems are the secret ingredient behind personalized online experiences and powerful decision-support tools in retail, entertainment, healthcare, finance, and other industries.

Recommender systems work by understanding the preferences, previous decisions, and other characteristics of many people. For example, recommenders can predict the types of movies an individual will enjoy based on the movies they've previously watched.

The three key objects managed by recommender systems are users, items and user-item interactions. These objects are tightly connected with each other and influence each other via various relations. For this very reason, recommender systems can be most naturally modelled by means of graphs through which the complex and heterogeneous nature of the available amount of information and data can be captured. It is therefore not surprising that in recent years the integration of graphs into recommender systems has attracted considerable attention from researchers and practitioners.

In a graph, the nodes correspond to entities (users and items), and edges correspond to relations between entities. Entities and their attributes can be mapped into a graph to understand the mutual relations between them.

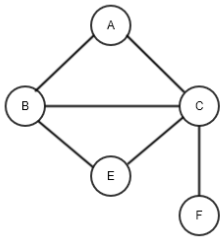
As a graph learning technique Graph Neural Networks (GNN) will be applied. GNNs have recently become very popular due to their ability to learn complex systems of relations or interactions arising in a broad spectrum of problems. They have proven to be among the best performing architectures for a variety of graph learning tasks. The key idea in GNNs is to learn how to iteratively aggregate feature information from local graph neighborhoods using neural networks. This aggregation step allows each node to learn a more general node representation from its local neighborhood.

Basic Theory

The basic elements of each graph are nodes and edges. **Nodes** can represent a variety of different objects such as persons, items, places and many more. **Edges** show how the nodes in the graph are connected. A simple graph consists of nodes and edges of only one type. Let's say the nodes represent persons and the edges show whether person A knows person B.

Edges can be **directed** or **undirected**. Person A might know person B, but person B has never heard of person A – that's a directed edge then. An undirected edge could be "married". If person A is married to person B, person B is automatically married to person A – at least in Switzerland. As we can see from these examples some edge types are naturally directed and others undirected. However, any directed graph can be transformed into an undirected graph by adding reverse edged.

Figure 1 shows a simple graph with nodes A to F. These nodes are connected by undirected edges. Let's stick to the example with persons who know each other. Person A knows person B and C, person B and C know person A and others. These relationships can be converted into an **adjacency matrix**, where each row and each column represents a person. The matrix values indicate whether an edge between two nodes exist or not.



Adjacency Matrix

	A	B	C	E	F
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	1
E	0	1	1	0	0
F	0	0	1	0	0

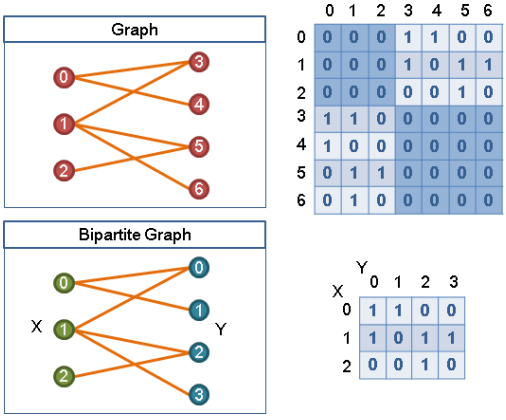
Figure 1: Simple Graph with adjacency matrix. Source: https://www.oreilly.com/

If the matrix values are boolean, we call it an **unweighted** graph. As we can imagine, edges could have a numerical value. If we think about cities and the flight distances between those cities. An edge could hold the information about how long the distance is. In the adjacency matrix we then would see numerical values instead of booleans – this then would be a **weighted** graph.

Nodes as well as edges can have additional attributes**.** For example, a person node could hold information about age, gender, or occupation of a person or an '*married_to*' edge could contain the marriage date, place etc. These **node and edge features** are represented in a vector. Vectors of the same node or edge type must be equal in their dimensionality.

Bipartite Graph

Bipartite graphs are a special form of graphs and a natural way to model users, items, and the interaction between those two. It is a common way to model recommender systems as a bipartite graph. As in Figure 2 shown, we have different types of nodes. On the graphs left-hand side, we see nodes of type X (green) and on the right-hand side we see node types Y (blue). Let's replace X with users and Y with items or even more specific with movies. The edges between users and movies could refer to *"has watched"*, *"rated"* or *"liked"*.



	0	1	2	3	4	5	6
0	0	0	0	1	1	0	0
1	0	0	0	1	0	1	1
2	0	0	0	0	0	1	0
3	1	1	0	0	0	0	0
4	1	0	0	0	0	0	0
5	0	1	1	0	0	0	0
6	0	1	0	0	0	0	0

X \ Y	0	1	2	3
0	1	1	0	0
1	1	0	1	1
2	0	0	1	0

Figure 2: Bipartite graph and interaction matrix. Source : https://web.ntnu.edu.tw

A bipartite graph cannot directly be converted into an adjacency matrix. Usually, a bipartite graph is represented by an interaction matrix. However, an interaction matrix can be transformed into an adjacency matrix, as it is shown in Figure 2. In context of GNNs and especially the PyG library, this is something to be aware of when working with bipartite graph structures.

Properties of Graph Data

Graph data is different to most other data structures we know in machine learning. Two properties which sets graph data apart from images or tabular data are:

- **Arbitrary size and shape**: It might be argued that this is also true for image data, for example. But images can simply be resized, padded, or cropped to the same size. Such operations are not defined on graph data. Additional nodes or edges cannot be removed. Therefore, methods are required which can handle arbitrary input sizes and shapes.
- **Permutation invariance:** Graphs that look differently can still be structurally identical. If an image is flipped the result is a different image. However, if a graph gets flipped only the order of nodes is different but its structure is still the same. Algorithms that handle graph data must be permutation invariant.

Recommender Systems

A recommender system is a system that tries to predict the value a user would grant to a certain item. Early recommender systems were mostly based on the content. They focused on the understanding of the product itself instead of knowing the user. These **content-based** filtering recommender systems elaborate a specific profile for each item and then calculate some similarity-based metric. It basically is about clustering similar items.

Later, **collaborative filtering** recommender system emerged. These systems focus on the user behaviour. The assumption behind it is, that users with a similar observation history tend to be interested in the same items.

Graph databases such as **Neo4j** make content-based and collaborative filtering quite easy. Relatively simple cypher queries get the job done. Some examples can be found in Neo4j's sandbox project for recommender systems or in this project repository on GitHub.

Graph Neural Networks (GNN)

In recent years GNNs have emerged and established themselves as state-of-the-art models in many fields – including recommender systems. GNNs can learn suitable representations of graph data to solve many graph problems such as label classification, link prediction or graph-level prediction.

Message passing layers (MPL) build the core of each GNN. A MPL gathers information about the neighbourhood of a node, aggregates this information and updates the current node embedding with the new information. This approach is also referred to as graph convolution and can be seen as an extension of convolutions on graph data.

Figure 3 shows on the left-hand side an image convolution and a graph convolution on the right-hand side. For images we can slide a learnable kernel over the grid structure of an image, which extracts then the most important information. This can also be seen as combining the information from a neighbourhood in a local area.

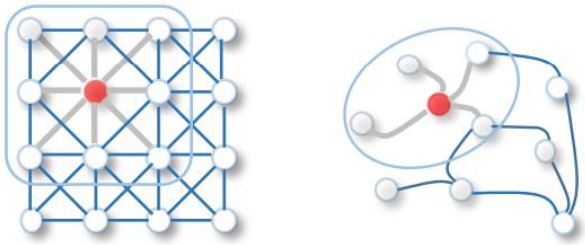


Figure 3: 2D Convolution vs. Graph Convolution. Source: Wu et al., 2019, A comprehensive Survey on Graph Neural Networks, https://arxiv.org/pdf/1901.00596.pdf

For graph convolutions this idea is extended. Instead of kernels we simply combine the information of neighbouring nodes (and edges) and create new node embeddings including the information about the neighbourhood.

Figure 4 illustrates how message passing works. In this example graph we have a yellow node, three blue nodes and a green node. After one message passing layer, the yellow node embedding includes information about its blue neighbours. After a second MPL the embedding of node one also holds information about the green node, or in other words it holds information about its neighbours' neighbours. This knowledge is stored in the node embeddings. These embeddings contain knowledge about the structure of the graph and about the node features.

Thus, the more message passing layers, the larger the neighbourhood. The number of MPL in a model is an important hyperparameter and must be chosen carefully. It depends on the learning task and on the graph data whether a rather small or large neighbourhood is relevant. A known problem is ***over smoothing***. In the example of Figure 4 we see, that after two MPL *Node 1* knows something about all other nodes in the graph. More layers would lead to over smoothing and result in bad node embeddings.

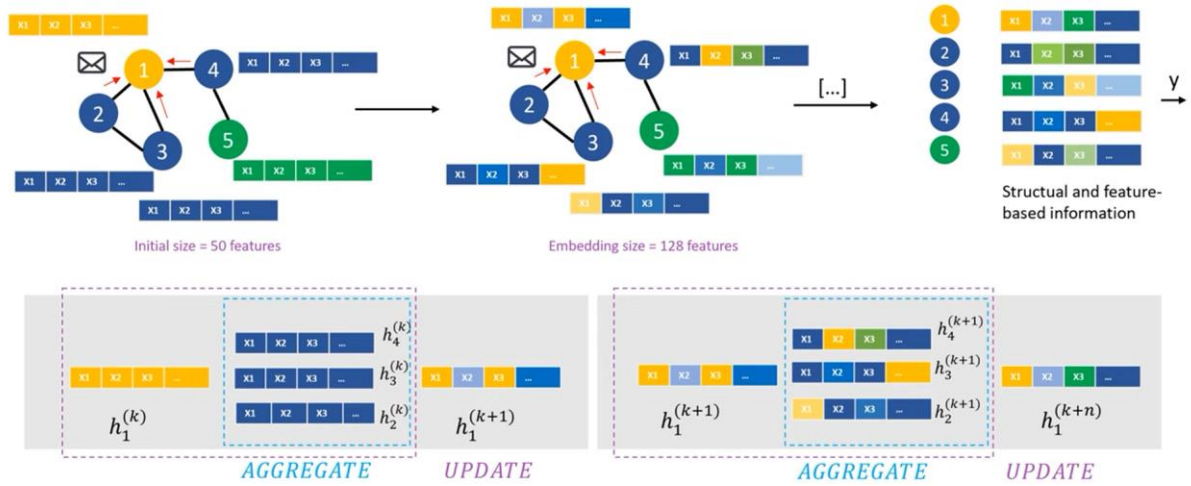


Figure 4: Illustration of Message Passing.
Source: DeepFinr, Understanding Graph Neural Networks | Part 2/3, Youtube

Dataset

MovieLens website is non-commercial and advertisement free website, which helps users to find movies they like. The website is run by GroupLens, a research lab at the University of Minnesota. GroupLens Research has collected and made available rating data sets from the MovieLens website. The datasets were collected over various periods of time, depending on the size of the set.

The datasets come in different sizes and flavours. The biggest dataset contains 25 million recommendations and the smallest 100 thousand. Some datasets contain additional information about movies (e.g. duration, release year, budget etc.) or demographical information about the users (e.g. age, profession etc).

Part of the project was also to familiarize with graph databases. Probably the most well-known graph database is Neo4j. Neo4j offers sandbox projects and one of these projects represents the MovieLens dataset. Hence, we decided to use the MovieLens data from the Neo4j sandbox project.

Figure 5 describes the MovieLens graph structure in Neo4j. The graph consists of users, who rated movies. The relationship *rating* has properties such as a timestamp when the user rated the movie and the rating itself. The worst rating is one and the best rating is five. Movies are assigned to genres (e.g., Comedy, Adventure, Thriller, Action etc.). Movies also have relationships to actors and directors.

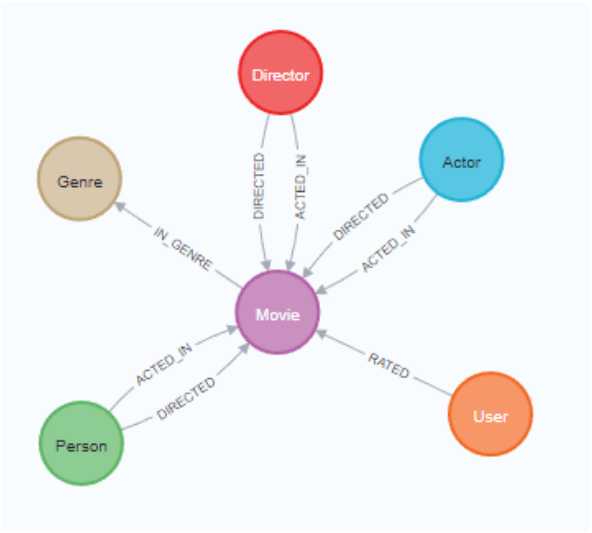


Figure 5: MovieLens Graph in Neo4j.

All properties are set to strings. Therefore, type castings for numerical and date values had to be executed. This allows calculations and easier comparison in cypher queries.

Table 1 summarizes the MovieLens graph with total number of instances, relevant properties and general remarks by type and label.

Type	Label	# Instances	Relevant properties	Remarks
Node	User	671	-	-
Node	Movie	9'125	Title, Budget, IMDB-Rating, plot, runtime, revenue, year	Many missing property values
Node	Genre	20	Name	-
Node	Actor	15'443	Bio, birthdate	-
Node	Director	4'091	Bio, birthdate	-
Node	Person	19'047	Bio, birthdate	-
Relationship	RATED	100'004	Rating	-
Relationship	ACTED_IN	35'910	-	-
Relationship	DIRECTED	10'007	-	-
Relationship	IN_GENRE	20'340	-	-

Table 1 - MovieLens Graph summary.

My MovieLens Recommender System

The project goal is to build a solid recommendation system on basis of the MovieLens dataset. The technology stack I used for this project consists of Neo4j as database and Python for building a GNN model. More specific, I used the PyTorch Geometric (PyG) library to build and train a GNN model. PyG is built upon PyTorch and consists of various methods for deep learning on graphs, from a variety of published papers.

To read data from Neo4j and load it into python objects, I used Neo4j's official python driver, which easily can be installed with pip. Furthermore, I used Python's classic data science stack with: Numpy, Pandas, Matplotlib, Scikit-learn etc.

Data pre-processing

As already mentioned before, some data pre-processing was executed directly in the database. These were mainly type castings for numerical and date types. But since the data must be modelled as a bipartite graph with only user and movie nodes, it was necessary to encode information about genres, actors, and directors directly on the movie node. Thus, one-hot encodings of genres were calculated and stored as arrays directly on the movie nodes.

To embed actor/director information on the movie, I have decided to consider only the most popular actors and directors. The assumption is that only famous actors will have a significant influence on user's movie ratings. The Cypher scripts used for the pre-processing can be found on GitHub.

PyG nodes are represented by fixed size vectors. Each dimension in this vector represents a feature. As an example, we could represent a movie by a four-dimensional vector, where the dimensions represents budget, runtime, release year, and revenue.

Users are represented in the same way. However, for users we do not have any relevant features. Hence, we have two options for reasonable user representations. The first would be to one-hot encode the users. With only 671 users in our case, that is a feasible solution but in a setup with way more users, one-hot encodings might be to sparse or even infeasible because of the enormous vector size. The second possibility is to find a good user embedding and use these embeddings as vector representations for users.

Figure 6 shows the comparison of pre-trained user embeddings versus user one-hot encodings. It shows on the lefthand side the loss and on the righthand side the training times. The results are based on a tenfold cross validation, where the same SAGE-Model was trained for 300 epochs. Although, the one-hot encodings performed on average slightly better, I will use the pre-trained embeddings because the training is much faster, and this approach is scalable.

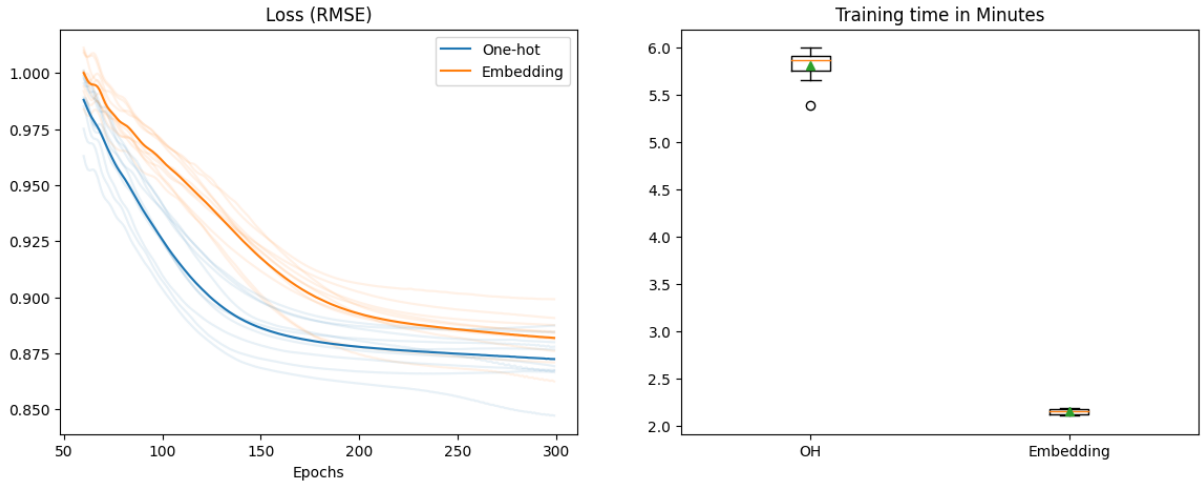


Figure 6: User Embeddings vs. One-hot Encodings.

During the project I experimented a lot with additional (movie) features. The dataset offers unstructured information such as a movie title and a plot, which is a very short description of the movie. The assumption was, that a movie plot could be very valuable for better predictions. Therefore, I transformed these information with a pre-trained hugging-face NLP-model into embeddings and used these embeddings also as features for the movie.

The hugging-face model embeds each sentence in a 384-demensional vector. I could use this vector directly and add it the as movie features. However, I was warried about very long training times and thought the plot might be overrepresented by 384 feature values compared to the 25 other movie features. Thus, I experimented with different dimensionality reduction methods and compared the results. Figure 7: Model training results with different plot embeddings.Figure 7 shows the learning curve of training the baseline SAGE model for 250 epochs with different plot embedding sizes.

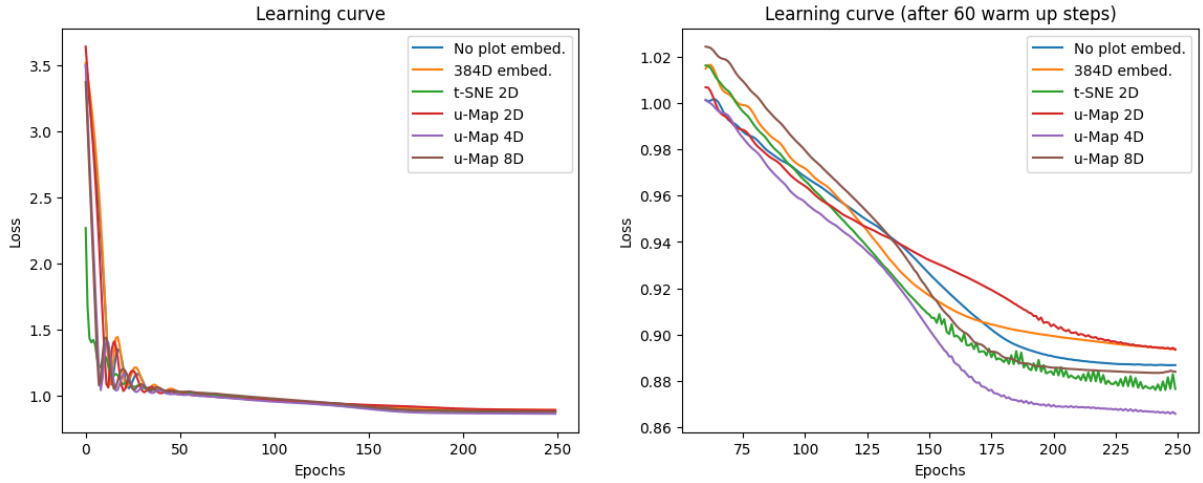


Figure 7: Model training results with different plot embeddings.

I ran the same test multiple times and the results were quite different each time. It is therefore hard to come up with useful conclusions. However, the u-Map 4D, the t-SNE 2D and no plot embeddings showed overall the best performances. I decided to use the u-Map reduced plot embeddings with four dimensions for the final model.

PyG processing

Depending on the chosen GNN model/layer, PyG expects a certain structure of the input graph. Most of PyG's models expect undirected and homogeneous graphs. Luckily, the library offers utility functions to transform directed into undirected graphs by adding reverse edges.

Moreover, PyG offers data classes for both homogeneous and heterogeneous graphs. The final PyG structure of our user-movie graphs looks as follows:

```
HeteroData(
user={ x=[671, 8] },
movie={ x=[9125, 29] },
(user, rates, movie)={
edge_index=[2, 100004],
edge_label=[100004]
},
(movie, rev_rates, user)={ edge_index=[2, 100004] }
)
```

The *HeteroData* class represents nodes and edges of different types in a python *dict* style. Users are represented by a 671x8 tensor. In other words, each user is defined by a pre-trained eight-dimensional vector. Each of the 9125 movies is represented by 29-dimensional feature vector.

The edges between users and movies are defined as *sparse tensor*. Instead of creating a huge and very sparse adjacency matrix – in our case the matrix would have 9796 by 9796 dimension – PyG expects a two-dimensional vector the first dimension contains the index of users and the second dimension the index of movies. The *edge label* holds the rating scores. The reverse edge type *(movie, rev_rates, users)* was created by calling a PyG utility function. This is necessary to transform a directed into an undirected graph.

PyG offers also functionality to split graphs into training, validation and testing graphs. I used the *RandomLinkSplit* class, which samples edges into sets of training, validation and test edges:

```
train_data, val_data, test_data = RandomLinkSplit(
    num_val=0.1,
    num_test=0.1,
    neg_sampling_ratio=0.0,
    edge_types=[('user', 'rates', 'movie')],
    rev_edge_types=[('movie', 'rev_rates', 'user')],
)(data)
```

80% of all edges were used for training, 10% for validation and the remaining 10% for testing. At this stage the data is ready to get trained on a PyG model.

Build and train a GNN model

In recent years graph neural networks (GNN) has been a very active research field and many new methods and models were published. During the project I have focused on two recently published papers. The first one is from Donghan Ye et al. (2021), in which they explain *their Knowledge Embedding Based Graph Convolutional Network*. The authors have focused on graphs with multiple heterogeneous relations. That could be for example, *has_viewed, added_to_cart, bought* etc. The MovieLens graph has only one relation type *rated*. Hence, we could not really test the benefits of the model promised by their authors.

The second paper in focus is from Rampasek et al. (2022) about a new approach for attention based GNNs. One of the biggest problems with attention-based networks is the quadratic computational

complexity. Rampasek et al. introduce the “GraphGPS” framework to enable general, powerful, and scalable graph transformers with linear complexity. The paper looked very promising, but it is not yet ready to be fitted to any *PyG HeteroData* class.

I have talked to Mr. Rampasek and asked him whether it is possible to use the **GraphGPS** model for the MovieLens link-prediction task. Unfortunately, GraphGPS does not support this setup. According to Mr. Rampasek the model expects an inductive learning task from a set of training graphs to a set of test graphs. He further stated that he is not aware of any recommendation systems based on graph transformers at the moment. Recommendation systems is an area he does not know much about, beyond PinSAGE-type models.

I guess this is the difference between graphs and for example images or tabular data. Graphs can have different shapes, sizes, and structures. There is no “one-model fits it all”. Just from these two papers I have noticed that the first model (Donghan Ye et al.) expects a graph with multiple relation types and the one from Rampasek et al. expects a set of graphs. I do not even know if these graphs must be homogeneous or not.

So, I focused on GNN models/layers, which were available in the PyG library. Luckily, PyG offers a quite comprehensive cheat sheet, which allows to quickly filter, what type of layer might be appropriate for your graph type. Filtering by *bipartite* and *edge_weight* just a few network operators remain.

After testing many of the operators, two of them have shown quite solid performance. By far the best results were achieved with the **SAGEConv** operator, significantly worse but compared to the datasets benchmarks (next chapter) still solid results have been achieved with the **GATConv** operator.

The model architecture consists of an encoder and a decoder part. The encoder consists of two message passing layers (e.g. SAGEConv, GATConv). Two fully connected linear layers make up the decoder part. Figure 8 shows the models loss (RMSE) on the left and the average training time on the right.

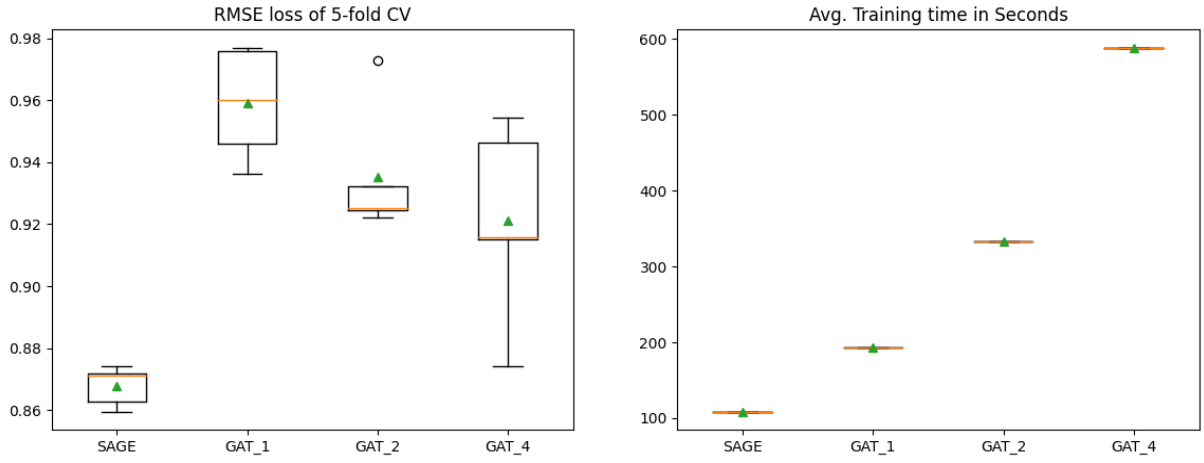


Figure 8: Model loss (RMSE) and training times

I have compared one SAGEConv and three GATConv models with an expensive fivefold cross validation. The convolution layers have 64 hidden channels and for activation I used the ReLu function. The GAT-models have an additional hyperparameter *number of attention heads*. I tested the same model with three different attention head numbers. These were the tested models:

- SAGE: Input -> [SAGEConv (64) -> ReLu] -> [SAGEConv (64)] -> [FC (128) -> ReLu] -> FC (1)
- GAT_1: Input -> [GATConv (64) -> ReLu] * 1 -> [GATConv (64)] * 1 -> [FC (128) -> ReLu] -> FC (1)
- GAT_2: Input -> [GATConv (64) -> ReLu] * 2 -> [GATConv (64)] * 2 -> [FC (256) -> ReLu] -> FC (1)
- GAT_4: Input -> [GATConv (64) -> ReLu] * 4 -> [GATConv (64)] * 4 -> [FC (512) -> ReLu] -> FC (1)

The SAGE-Model achieved the best average loss (RMSE of ~ 0.87) and with an average training time of around 100 seconds it was by far the fastest too. This comparison of different convolution types can be seen as a preselection for the final fine-tuning of the best model.

For hyperparameter optimization of the SAGE-Model, I have tested different settings of these hpyerparameters:

- Number of epochs
- Model size and architecture
- Aggregation type and activation functions
- Regularization methods

Reducing the model capacity led to better results because the model could not overfit after a few epochs. With a smaller model I was able to train for 600 epochs without suffering from overfitting. The final SAGE-Model looks as follows:

```
<Model(
  (encoder): GraphModule(
    (conv1): Module(
      (user__rates__movie): SAGEConv((-1, -1), 24, aggr=mean)
      (movie__rev_rates__user): SAGEConv((-1, -1), 24, aggr=mean)
    )
    (conv2): Module(
      (user__rates__movie): SAGEConv((-1, -1), 24, aggr=mean)
      (movie__rev_rates__user): SAGEConv((-1, -1), 24, aggr=mean)
    )
  )
  (decoder): EdgeDecoder(
    (lin1): Linear(in_features=48, out_features=24, bias=True)
    (dropout): Dropout(p=0.0, inplace=False)
    (lin_out): Linear(in_features=24, out_features=1, bias=True)
  )
)>
```

Evaluation

On paperswithcode I found some RMSE benchmarks for the MovieLens 100K dataset. The best performing models are GLocal-K with a RMSE of 0.889 and GHRS with a RMSE of 0.887. Figure 9 shows the models and their RMSE loss on the MovieLens 100K dataset.

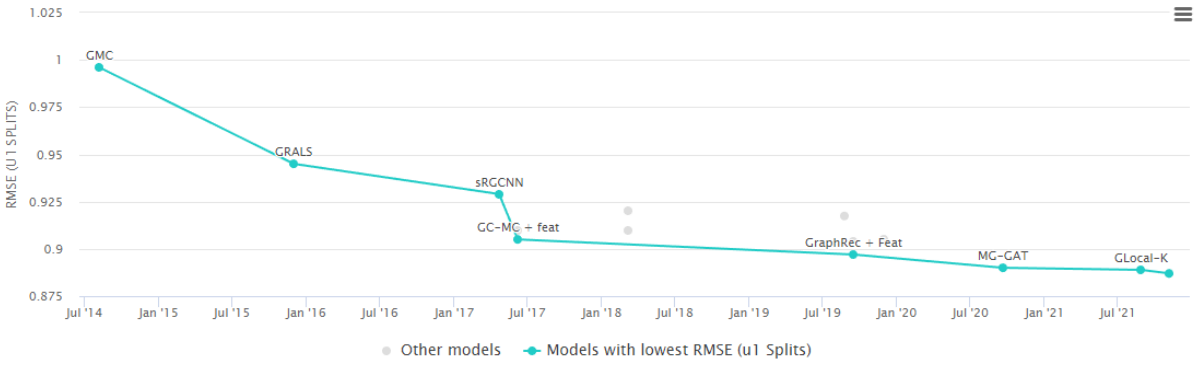


Figure 9: Benchmark recommendation systems on MovieLens 100K.

My SAGE-Model beats these benchmarks. This was very surprising for me. I figured out that the dataset used for GLocal-K and GHRS do not have any features. The datasets they used only contain User-ID, Movie-ID, and the rating scores.

Figure 10 shows the learning curve of my SAGE-Model as well as the GHRS's RMSE as reference point.

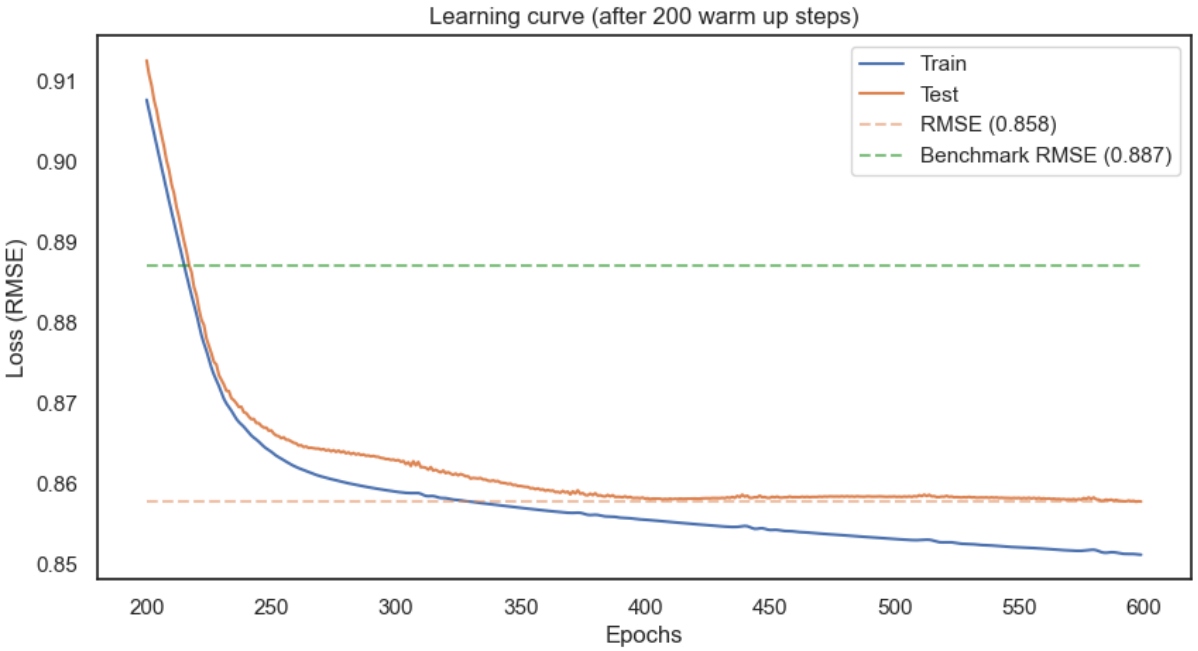


Figure 10: Learning curve of final model.

The RMSE is well suited to compare it with benchmarks. However, RMSE is not the best metric to evaluate whether the model makes good recommendations. More suitable metrics are **precision@k** and **recall@k**. These are modified versions of the binary precision and recall metric for evaluating recommendation systems.

First, we must define a **threshold** to classify our recommendations into "*relevant*" and "*not relevant*". With a threshold of 3.5, ratings greater or equal the 3.5 are *relevant* and ratings below the 3.5 are *not relevant*.

In the context of recommendation systems we are most likely interested in recommending top-N items to the user. So it makes more sense to compute precision and recall metrics in the first N items instead of all the items. Thus **k** defines how many items we want to recommend to the user.

Precision@k is the proportion of recommended items in the top-k set that are relevant. Suppose that my precision@10 is 80%. This means that 80% of the recommendation I make are relevant to the user.

Recall@k is the proportion of relevant items found in the top-k recommendations. Suppose that we computed recall@10 and found it is 40%. This means that 40% of the total number of the relevant items appear in the top-k results.

Table 2 shows the results of the final model. We want to recommend 20 items therefore k=20. For the threshold I chose to values (3.5 and 4.0) for better evaluation.

	Threshold=3.5	**Threshold=4.0**
Precision@20	0.7238	0.3782
Recall@20	0.6644	0.3486

Table 2: Precision@k and Recall@k

Precision@20: 38% of the model's recommendations are relevant to the user if "relevant" is defined as movies rated with 4.0 or higher. 72% of the model's recommendations are relevant to the user if "relevant" is defined as movies rated with 3.5 or higher.

Recall@20: On average 35% respectively 66% of an users top relevant movies are recommended by the model.

Conclusion

Overall, I am very happy how well the GNN performed on the movie recommendation task. To be honest I was surprised to see that the SAGE-Model beat the benchmarks on paperswithcode.com. However, an enriched datasets with information about genres, actors and directors is very valuable and probably the main reason for outperforming the SOTA benchmarks.

During model training I have noticed that the training is kind of unstable. Sometimes after 300 epochs the test loss got stock at around 0.93. The next training run then performed well again. That is something to improve. Probably tweaking the learning rate – maybe with a more appropriate schedule – would help.

I used *Root Mean Squared Error (RMSE)* as loss function mainly for two reasons: first it was kind of natural due to the rating-schema (1-5) and second to compare the results with SOTA benchmarks. However, it would be interesting to see if precision@k and recall@k could be improved by using another loss function. As an example, the rating could be binarized and only edges with a positive value could be used. The relations in the graph would then change from “rated” to “liked”. With negative sampling a model could be trained on a cross-entropy loss or a rank-based loss function. That might lead to better precision@k and recall@k values.

Finally, the inner workings of PyG’s implementation of the SAGEConv layer is still kind of a black box to me. It would be very nice but time consuming as well, to fully understand how the message passing in the SAGEConv layer works.