ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Recommender Systems

GRAPH NEURAL NETWORKS FOR RECOMMENDER SYSTEMS

CANDIDATE
Oleksandr Olmucci Poddubnyy

SUPERVISOR
Prof. Paolo Torroni

CO-SUPERVISOR
PhD. Piotr Bigaj

Academic year 2020/21

Session 3rd

To everyone who has supported me.

Contents

List of Figures

List of Tables

Abstract

In recent years, a new type of deep learning models, Graph Neural Networks (GNNs), have demonstrated to be a powerful learning paradigm when applied to problems that can be described via graph data, due to their natural ability to integrate representations across nodes that are connected via some topological structure. One of such domains is Recommendation Systems, the majority of whose data can be naturally represented via graphs. For example, typical item recommendation datasets can be represented via user-item bipartite graphs, social recommendation datasets by social networks, and so on. The successful application of GNNs to the field of recommendation, is demonstrated by the state of the art results achieved on various datasets, making GNNs extremely appealing in this domain, also from a commercial perspective. However, the introduction of graph layers and their associated sampling techniques significantly affects the nature of the calculations that need to be performed on GPUs, the main computational accelerator used nowadays: something that hasn't been investigated so far by any of the architectures in the recommendation literature. This thesis aims to fill this gap by conducting the first systematic empirical investigation of GNN-based architectures for recommender systems, focusing on their multi-GPU scalability and precision speed-up properties, when using different types of hardware.

Chapter 1

Introduction

Recommendation Systems, also known as Recommender Systems is a large subfield of information technology that deals with software systems that filter or propose content to users, based on various properties, such as their historical preferences, items similar to what they are currently viewing or listening, items that other similar users have liked and many more.

We are exposed to recommendations in almost every major service we use in our daily life. For example, when watching a video on a video hosting platform, we are promptly presented with a collection of new videos, similar to what we either like or have watched recently. When navigating e-commerce websites, we are presented with suggestions about what could be the item we are interested in purchasing next or what are the items that go well together with the currently viewed item. On social media we are recommended which people might be our potential friends, or suggested influencers to follow, that promote the topics which we might be interested in, and so on.

In recent years, Deep Learning, a subfield of Artificial Intelligence has developed methods that have greatly benefitted a variety of fields, among which Computer Vision and Natural Language Processing. Many tasks of Computer Vision did see a significant accuracy improvement when using more sophisticated convolutional architectures that greatly exploited the grid nature of images and videos [51][44][41]. A similar phenomenon has also occurred in Natural Language Processing, where architectures based on transformers, trained on massive corpora of textual data have shown to be more accurate on majority of the tasks addressed by the field, such as sentence classification, named entity recognition, questions answering and many more [56][12][32].

The field of Recommendion Systems has also benefited from the advances in Deep Learning. In fact, traditional recommendation methods have evolved from being simple matrix-based methods, able to only capture simple linear interactions between users and items, to include complex, deep interaction models, aimed to capture higher order and more sophisticated interactions. Despite the progress, both academic and many commercial deep recommendation methods continued to rely on the same, simplistic type of datasets, composed of <user, item, feedback> triples, due to their simpler availability. It is however worth noting that deep architectures generally support the inclusion of other modalities of information, such as content information, that could be used to boost recommendation accuracy when such data is available.

Further advancement in Deep Learning, led to a new type of neural networks, called Graph Neural Networks (GNNs) to emerge. The main motivation behind GNNs is that they allow to perform learning over arbitrary input data topology. This can be viewed as a generalization over more known approaches that are used to process grid-like data, such Convolutional Neural Networks in case of image data, and sequential models, used to process sequential data, such as text. The introduction of learning approaches that can work on arbitrary data topology has helped to achieve progress in multiple fields such as automatic drug discovery [52], molecule analysis [53], road traffic prediction [11], point clouds [15] and many more. It turns out that graph structures can be also used to naturally represent the data for the majority of recommendation tasks. In fact, GNNs have recently been successfully applied to the field of recommendation [54][60][70], beating state of the art deep learning approaches [20][33].

Despite their success, graph based approaches in recommendation have seldom been studied from the performance point of view, which is one of key points of success of the modern Deep Learning methods. Being able to efficiently train large models on big datasets, requires both specialized hardware accelerators and optimized software to exploit their capabilities. One of the most popular and used accelerators are NVIDIA GPUs, which were originally used as specialized accelerators for computer graphics, supporting, via hardware, mathematical calculations related to graphics, as well as offering an API, called CUDA, to customize the existing functionalities. Thanks to CUDA, it became possible in the earlier days of modern Deep Learning to take advantage of GPUs, by exploiting the optimized linear algebra routines, that in their turn allowed to perform parallel computation on large blocks of data, enabling us to train Deep Learning models on this type of hardware [28].

Motivations behind the work

This work can be viewed as a commercially motivated, exploratory work. The commercial motivation behind the work, is given by the fact that NVIDIA, in particular the engineering team of which the thesis author is a part of, is interested in implementing highly-optimized, state of the art deep learning models in the recommendation domain, for their Github examples repository[1]. The importance of this repository comes from containing and maintaining open source, reference implementations of models that are both state of the art in their domains and commercially attractive for industrial clients. The latter is achieved thanks to the models being highly optimized, “plug-and-play” examples as well as them being implemented to better use all of the features of newest NVIDIA GPUs such as multi-node training, Automatic Mixed Precision (AMP), TensorFloat-32 math mode, XLA and others.

[1] `https://github.com/NVIDIA/DeepLearningExamples`

Implementing a model in a scenario with such a high degree of optimization, requires a certain level of cooperation between various teams (e.g. natively implementing certain operators into frameworks or solving critical lower-level bugs that have negative impact on the model's performance) and months of full-time work devoted to testing, profiling and polishing of the model, to better exploit the available computational resources, on various industrial hardware setups (e.g. DGX-V100 or DGX-A100). The previous statement suggests that in order to achieve commercial success, when implementing a model, the model itself needs, among many other properties, to offer good performance properties in terms of:

1. **GPU utilization**: the model needs to utilize a single GPU as efficiently as possible to justify the customer's investment into better hardware.
2. **Multi-GPU scaling**: the model needs to scale well when increasing the amount of computational resources. This scaling can be both used to perform training of larger models, or to train the model faster on a fixed amount of data.
3. **Math modes speed-up**: the model needs to provide performance speed-ups when a less precise, but more compact math mode is being used (e.g. Automatic Mixed Precision or TensorFloat-32 modes).

which requires a thoughtful literature review, analysis and planning ahead of time, on which model could potentially satisfy the above criteria, before even starting its costly implementation. When working with traditional Deep Learning models, such procedure can be simplified, as generally, those types of models have been studied and implemented for a longer period of time, yielding both efficient reference implementations and primitives to build them in existing frameworks. On the other hand, when working with GNNs, the novelty of the graph based layers themselves requires new tooling to be used, in terms of libraries and frameworks, which implies that some of the features described in state of the art literature might be either inefficiently, wrongly or not implemented at all. The latter, plus the fact that graph models in recommendation tend to be shallow, suggests that when studying performance properties, we are to prefer simpler models, even toy ones, since they can be used as a proxy to study the performance of more complex models, without the actual implementation burden, such as custom layers, mixed code quality and usage of wrong technologies.

Goals

Considering the previous motivations, the goals this work tries to achieve are:

1. To identify and describe a pool of state of the art graph recommender models, that have an existing code implementation, provided by their authors.
2. To characterize the models based on model properties such as model size, sampling and type of data used, that could possibly affect their performance.
3. To empirically study the performance properties on a simplified GNN model, that offers similar model properties to the analyzed state of the art models.

In the light of these goals, we will additionally perform a critical analysis on the existing literature of the state of the art graph based models for recommendation systems.

Structure of the work

This work is organized in six chapters:

- **Chapter 1**: introduction of motivations and goals behind this work.
- **Chapter 2**: introduction of the recommendation systems field with its main tasks, datasets, evaluation metrics, models and open problems.

- **Chapter 3**: introduction of graph neural networks as an extension of deep learning over arbitrary structured inputs, description of main building blocks and graph based layers and how to make them work on larger graphs.
- **Chapter 4**: overview on how graph neural networks can be used to improve recommendation, describing state of the art models and characterizing them from the performance point of view.
- **Chapter 5**: description of the experimental setup.
- **Chapter 6**: concluding remarks.

Chapter 2

Recommender Systems

Recommender Systems, also known as Recommendation Systems is a subfield of information technology that deals with software systems that can filter or propose content to users based on their preferences.

Historically, the field of recommendation was based on simple matrix methods (e.g. matrix factorization) to solve the majority of its tasks. With the recent advancements in Deep Learning, recommendation models have evolved from being simple matrix-based methods used to capture simple interactions between users and items, to becoming more complex models that can capture higher order and more sophisticated interactions.

This chapter will be dedicated to describing the problem of recommendation in terms of possible recommendation task formulations, including common datasets, metrics and loss functions. We will then briefly cover different families of classical and early Deep Learning based recommenders. Finally we will see some open problems that are relevant in this field, namely: cold start, data sparsity and large embedding tables.

2.1 Introduction

A recommender system could be viewed as a function that given a user, a set of items and other contextual information in input, would output a ranked list of items that the user would most likely consume. The task of recommendation itself consists in finding relevant items in an existing collection (e.g. database of items) and ranking them based on a certain user's objective (e.g. click, purchase).

In this section we will first introduce the concept of user feedback and some recommendation tasks, focusing on those that we will encounter in this work. We'll then have an overview of commonly used recommendation datasets, loss functions that are used in personalized recommendation and finally the evaluation metrics that are typically used to assess the quality of a recommendation system.

2.1.1 User Feedback

Before describing the common recommendation tasks, it is important to define the concept of user feedback, which is a key concept of recommendation. When users perform actions over items on a platform (e.g. an e-commerce site, a movie review system or a video blogging platform and so on) they provide implicit or explicit feedback:

- **Implicit feedback**: implicitly represents user's preference over items by taking into consideration their behaviour on the platform, it does not require any proactive thinking by the user (e.g. user clicks an ad or visits a page). This type of feedback is generally of a binary nature.

- **Explicit feedback**: requires the user to proactively think before expressing their preference over an item (e.g. a user ranks a movie on a scale from 1 to 5 stars). This type of feedback is of a non-binary nature.

Many recommender systems are centered over implicit feedback as its easily obtainable, since it indirectly reflects user's opinion through their observed behaviour (e.g. by analyzing their browsing history, mouse movements, etc.). The problem with implicit feedback is however that it's inherently noisy. For example if a user has watched a movie, we can't necessarily indicate whether they have liked it or not. The explicit feedback on the other hand might be more informative over user's preferences, however it is not always readily available as many users might be reluctant to rate the items they consume.

2.1.2 Recommendation Tasks

Based on the available type of feedback we can define different types of recommendation tasks that can be approached by a recommendation system:

- **Click-through rate (CTR) prediction**: predicting the probability that a user will consume an item, characterized by a set of features (e.g. image, text, day of week, position, user features, etc.), namely predicting $\mathbb{P}(\text{click}|\text{item, user, features})$ of an implicit feedback occurring.
- **Rating prediction**: predicting the probability that a user would assign a certain rating to an item, characterized by a set of features, namely predicting the $\mathbb{P}(\text{Rating=}r|\text{item, user, features})$ of assigning, an explicit feedback, rating r to a given item.
- **Sequential prediction**: predicting the probability distribution of the next target item consumed by an user, based on a *sequence* of previously consumed items, namely $\mathbb{P}(\text{Target}|\text{sequence, features})$. Both user and item sequence could be characterized by an additional set of features, besides their ids. If user features are not included, the problem becomes a generalized (rather than personalized) recommendation.

Side features

As seen from the tasks formulation, the probability of consumption or of ranking is characterized by both the information about user and item in terms of their ids as well as the features that represent their properties such as age, gender, income and so on for users, and price, color, type, picture, description and so on for items. This last type of user and item features is called **side features** (or side information), and as we will see in Section 2.2, families of models differ by the type of information they use. Content based models rely on side-information to make the predictions. Collaborative filtering models make use only of user and item ids in order to make predictions. Hybrid models on the other hand, combine both ids and side features when estimating the user's feedback.

2.1.3 Common datasets

MovieLens

The *MovieLens* [1] dataset is probably one of the most popular datasets that is available for recommendation research. The dataset itself was gathered by the homonymous non-commercial web-based platform used for movie recommendation, created in 1997. MovieLens data is composed of user, movies and the ratings users gave to movies, and is available in several versions, that vary based on the amount of available ratings on a scale from 1 to 5, namely:

- **MovieLens 100K**: 943 users, 1682 movies and 100000 ratings.
- **MovieLens 1M**: 6040 users, 3706 movies and 1000209 ratings.
- **MovieLens 10M**: 69878 users, 10677 movies and 10 million ratings.
- **MovieLens 25M**: 162000 users, 62000 movies and 25 million ratings.
- **MovieLens latest-full**: the most recent version (at the time of writing) that includes 280000 users, 58000 movies and 27 million ratings.
- **MovieLens 1B**: synthetically generated version of MovieLens that includes 1 billion ratings.

Category	Users	Items	Ratings
Books	8,201,127	1,606,219	25,875,237
Cell Phones & Accessories	2,296,534	223,680	5,929,668
Clothing, Shoes & Jewelry	3,260,278	773,465	25,361,968
Digital Music	490,058	91,236	950,621
Electronics	4,248,431	305,029	11,355,142
Grocery & Gourmet Food	774,095	120,774	1,997,599
Home & Kitchen	2,541,693	282,779	6,543,736
Movies & TV	2,114,748	150,334	6,174,098
Musical Instruments	353,983	65,588	596,095
Office Products	919,512	94,820	1,514,235
Toys & Games	1,352,110	259,290	2,386,102
Total	20,980,320	5,933,184	143,663,229

Table 2.1: Category information for the Amazon-review dataset.

Amazon-review

Amazon-review dataset [35] [19] is another dataset that includes 11 years worth of user reviews and ratings for different categories of goods that was released by Amazon in 2016. The dataset consists of both information about user ratings, the reviews and the information about the products in question (e.g. description, picture). Product categories are unevenly balanced, resulting in a total of 20,980,320 users, 5,933,184 items and around 143 million ratings. A more detailed overview of the product categories present in the training set can be seen in Table 2.1.

Flixster

Flixster, first introduced in [22], is a dataset that was obtained by crawling a large-scale social network during a period from November 2005 to November 2009, from the homonymous website that allowed users to rate movies and add friends. Flixster has ratings that can take 10 discrete values in the interval from 0.5 to 5, with a step of 0.5. The statistics for the dataset are: 147,612 users, 48,794 items, 8,196,077 and 2,538,746 social connections. A common preprocessing of this dataset was given by Monti in [36], that uses 3,000 users,

[1] `https://grouplens.org/datasets/movielens/`

3,000 items and has 26,173 ratings.

Douban

Douban dataset, introduced in [34] was collected from the homonymous Chinese platform that allows users to provide ratings and reviews for movies, books and music on a discrete scale from 1 to 5 and to have friend connections. The dataset itself contains 129,490 unique users, 58,541 unique movies, 16,830,839 movie ratings and 1,692,952 friendship links between users. A common preprocessing of this dataset was given by Monti in [36], that uses 3,000 users, 3,000 items and 136,891 ratings.

YahooMusic

YahooMusic dataset, was first introduced in [13] and used as a challenge dataset for KDD-Cup 2011. The data itself was collected from the Yahoo! Music platform, over the span of 11 years, from 1999 till 2010. The challenge introduced two datasets:

1. **Track 1**: composed of 1,000,990 users, 624,961 music items and 262,810,175 ratings.
2. **Track 2**: composed of 249,012 users, 296,111 music items and 61,944,406 ratings.

where ratings were given on a scale from 0 to 100. The musical items include tracks, albums, artists and genres distributed in a non-uniform way (e.g. tracks are 81.15% of all musical items). Ratings are also not distributed in an uniform way (e.g. tracks are 46.85% of reviews).

A common preprocessing of this dataset was given by Monti in [36], that uses 3,000 users, 3,000 items and 5,335 ratings.

2.1.4 Loss functions

Binary loss

Binary Loss is a reformulation of a classification loss that can be used in case of problems which have implicit feedback. Given a set of positive items D^+ and a set of negative items D^-, the equation for binary loss is given by:

$$\text{Binary}(D^+, D^-) = -\frac{1}{|D^+|} \sum_{(u,i) \in D^+} \log(\sigma(\hat{y_{ui}})) - \frac{1}{|D^-|} \sum_{(u,j) \in D^-} \log(1 - \sigma(\hat{y_{uj}}))$$

where σ is a sigmoid function, defined as:

$$\sigma(x) = \frac{1}{1 + exp(-x)}$$

Such formulation of the loss, pushes the scores of positive interactions higher than those of negative ones. This is equivalent to training recall metric, as we're rewarding the model for recalling the positive interactions. The problem with such a formulation however is that *all the positive* interactions scores are pushed higher than those of *all the negative* scores, causing unnecessary loss penalization on the model's predictions even if the training recall metric is perfect. Another problem is also that this loss does not take into consideration the personalization factor. In fact, the positive and negative interactions are considered across multiple users at once, as it computes the sum for positive interactions separately from the negative interactions. The Bayesian Personalized Ranking Loss that we'll see next shows how to solve the before-mentioned problems.

Bayesian Personalized Ranking Loss

Bayesian Personalized Ranking (BPR) Loss [43] is a pairwise personalized ranking loss used by many recommendation models as an improvement over the binary loss. Data used by this loss consists of positive and negative pairs (items that user did not consume) of items, where the assumption is that user consumes positive items and does not consume negative ones, thus expresses lower to no preference for them. Formally, a training sample is a triple of form (u, i, j) which indicates that an user u likes an item i over an item j. The Bayesian formulation of BPR loss aims to maximize the posterior probability:

$$P(\Theta| >_u) \propto P(>_u |\Theta)P(\Theta)$$

where Θ represents parameters of an arbitrary model class and $>_u$ represents the desired personalized total ranking of all items for user u.

From this posterior probability formulation we can derive an optimization criterion using maximum posterior estimator:

$$\begin{aligned}
\text{BPR-OPT}(D) &= \ln P(\Theta| >_u) \\
&= \ln P(>_u |\Theta)P(\Theta) \\
&= \ln \prod_{(u,i,j)\in D} \sigma(\hat{y}_{ui} - \hat{y}_{uj})P(\Theta) \\
&= \sum_{(u,i,j)\in D} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) + \ln P(\Theta) \\
&= \sum_{(u,i,j)\in D} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) - \lambda_\Theta \|\Theta\|^2
\end{aligned}$$

where $D = \{(u, i, j)|i \in I_u^+ \wedge j \in I \setminus I_u^+\}$ represents the training set where I_u^+ denotes the set of items that user u consumed and $I \setminus I_u^+$ the set of all the items that user u did not consume. Additionally, $\hat{y}_{ui}$ and $\hat{y}_{uj}$ indicate the predicted scores of the user u towards items i and j respectively.

2.1.5 Evaluation metrics

RMSE

Root Mean Squared Error (RMSE) is a simple and widely used metric to evaluate the accuracy of a model that predicts the value of a continuous variable, such as an explicit feedback (e.g. a rating an user would give to a movie), for known user-item pairs:

$$\text{RMSE}(r, \hat{r}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(r_i - \hat{r}_i)^2}$$

Precision @ K and Recall @ K

We can evaluate a model that outputs continuous values using Root Mean Squared Error, however in case we would like to evaluate a model that produces binary output (e.g. relevant/non-relevant or clicked/not clicked), we need a change of metric to be more effective. Also, as we are dealing with a recommendation task, we are mostly interested in recommending top-N items to a user, instead of the entire items list in the system. Thus, the metric we want to employ should be parameterized by an integer K, which tells us how many items to take into consideration when computing the metric.

Given a set of top-K recommended items, **Precision @ K** is the proportion of the recommended items that are relevant:

$$\begin{aligned}\text{P}_u\text{@K} &= \frac{\text{TP}_u}{\text{TP}_u + \text{FP}_u} = \frac{\sum_{i=1}^{K} \text{rel}_{ui}}{K} \\ \text{Precision@K} &= \frac{\sum_{u=1}^{U} \text{P}_u @K}{K}\end{aligned}$$

we say that an item is *relevant* if the item was actually consumed in the ground-truth dataset.

On the other hand, **Recall @ K** is the proportion of relevant items found in the top-K recommendations:

$$\begin{aligned}\text{R}_u\text{@K} &= \frac{\text{TP}_u}{\text{TP}_u + \text{TN}_u} = \frac{\sum_{i=1}^{K} \text{rel}_{ui}}{|\text{rel}_u|} \\ \text{Recall@K} &= \frac{\sum_{u=1}^{U} \text{R}_u @K}{K}\end{aligned}$$

where $|\text{rel}_u|$ is the total number of consumed items by user u.

MAP @ K

Mean Average Precision (MAP) is another metric that's used to evaluate recommender systems that work on implicit feedback/binary output that takes the order of the results list into consideration. In order to compute it, we first define the concept of average precision.

Given a value K, average precision @ K (AP@K) metric represents the average of precision values for relevant items from 1 to K:

$$\mathrm{AP}_u@\mathrm{K} = \frac{\sum_{i=1}^{K} \mathrm{rel}_{ui} \mathrm{P}_u@i}{\sum_{i=1}^{K} \mathrm{rel}_{ui}}$$

the value of the metric is higher if the recommended items are both relevant and presented in higher positions of the results list.

The mean average precision at K (MAP@K) is given as the mean of AP_u@K across all the users in the dataset:

$$\mathrm{MAP@K} = \frac{\sum_{u=1}^{U} \mathrm{AP}_u@\mathrm{K}}{U}$$

NDCG@K

Normalized Discounted Cumulative Gain (NDCG) is a metric used to evaluate recommender systems with explicit feedback that allows to quantify the quality of current ordering of top-K relevant items vs a perfect ordering of top-K most relevant items.

In order to compute NDCG we first need to define the concept of *cumulative gain*, which corresponds to the sum of all the relevance scores (i.e. ratings given to items) up to K, of a top-K recommendations list:

$$\mathrm{CG@K} = \sum_{i=1}^{K} rel_i$$

Cumulative gain on its own does not take into consideration the ordering of the resulting items. An ordering discount term could be introduced as items at distant positions in a ranking are generally less influential than at earlier ones. This results in the *discounted cumulative gain* metric:

$$\text{DCG@K} = \sum_{i=1}^{K} \frac{rel_i}{\log_2(i+1)}$$

Finally, in order to compare recommenders that potentially return a different amount of results, we introduce a normalization term which equals to the discounted cumulative gain of a perfect order of top-K items, by relevance:

$$\text{IDCG@K} = \sum_{i=1}^{K} \frac{rel_i^{ideal}}{\log_2(i+1)}$$

Taking into consideration all of the previously defined terms, we can compute the *normalized discounted cumulative gain* using the following formula:

$$\text{NDCG@K} = \frac{\text{DCG@K}}{\text{IDCG@K}}$$

MRR @ K

Mean Reciprocal Rank (MRR) is a measure used to evaluate the quality of sequential recommenders, or generally the quality of a ranking system, based on the position of the correctly recommended items in a ranked list. A large MRR value indicates that correct recommendations are at the top of the ranking list.

The *reciprocal rank* (RR) of a list of recommendations provided to a user u, is defined as the inverse position (called rank) of the first relevant/consumed item among the first K items in the list:

$$\text{RR}_u\text{@K} = \frac{1}{\text{rank}_{u,1}}$$

For example, if we are considering a list of K=5 recommendations, assuming that there's a consumed item within the K recommendations, then depending on its position, the reciprocal rank can take the following values: $1/1, 1/2, 1/3, 1/4, 1/5$.

MRR @ K is the average of reciprocal ranks of the correctly-recommended items across all the users in the dataset:

$$\mathrm{MRR@K} = \frac{\sum_{u=1}^{U} \mathrm{RR}_u\mathrm{@K}}{U}$$

2.2 Families of Recommenders

The literature of recommender systems [2] is vast and is characterized by different academic and industrial approaches that can yield to complex categorizations and taxonomies. In this section we will give a bird-eye categorization over main families of recommenders. It is also worth noting that different authors might interpret different families in slightly different ways, in the subsequent categorizations we will try to provide just one single interpretation focusing on main differences.

2.2.1 Content Based Filtering

Content based filtering methods, as their name suggests, are methods that use similarities between items' or users' "contents", expressed by the means of their characteristics (e.g. side information or other features that are not a pure numerical identifier) to perform recommendation. They operate under the assumption that an user would more likely consume items similar to the items they have previously consumed. These methods generally try to model the function:

$$f(\text{user\_features}, \text{item\_features})$$

where user features can be for example age, gender, income and item features can be for example price and color.

Modern content based recommenders typically work by embedding items to a low-dimensional embedding space. Inside this low-dimensional space, we can use distance, comparison or similarity functions such as dot-product or

Euclidean distance to find closest items. Those approaches have been particularly successful in multimedia recommendations, in fact we'll mention some of those that can be viewed as content based recommenders.

Deep content based music recommendation [55] paper introduces an automatic music recommendation system that uses a convolutional neural network (CNN) to produce compact latent representations of mel spectrograms of user's liked songs, and suggests to the user songs that are similar to what the user has listened to.

General image retrieval models, such as the model in "Deep Metric Learning via Lifted Structured Feature Embedding" [48], DeepID [50] face recognizer, Facebook's e-commerce product recognizer Groknet [3], can be also viewed as a content based recommendation models, as they learn to embed images into a latent space on top of which multiple tasks can be solved, including item recommendation.

2.2.2 Collaborative Filtering

Collaborative filtering methods are a family of methods based on similarities between users. The concept of collaborative filtering was first described in [16], as an alternative to the existing content based filtering paradigm, where the authors proposed an effective approach to filter documents in a platform by the means of user collaboration. The system would record the user's reactions to the documents they read and use this information to help other users to filter particularly interesting or uninteresting documents.

Those methods generally describe both users and items via their ids and operate under the assumption that users who are similar in their behaviors would also exhibit similar preferences over items. For example, in a movie recommendation scenario, when making a prediction about how a user u would rate a movie m (rating unknown at the time of query), a collaborative recommender system would first try to find users with similar tastes in movies to user u (i.e. they rated movies in a similar way), and then try to estimate the rating based on ratings of its peers. More generally, when trying to predict the consumption of an item s by a given user u, the likelihood of consuming it is estimated based on consumption likelihood for u_j, users similar to u.

Collaborative filtering models based on learning, model the function:

$$f(\text{user\_id, item\_id})$$

and are composed of two main parts:

1. **Embedding**: users and items are first embedded, via a learnable embedding transformation, to latent representations, where they could be better compared.

2. **Interaction modeling**: historical interactions are reconstructed from latent representations of users and items, via some interaction function, which in its turn could be learnable (e.g. a multi layer perceptron) or not (e.g. an inner product).

We will start our discussion about collaborative filtering models with classical approaches, such as Matrix Factorization (MF), Factorization Machines (FM) and Sparse Linear Methods (SLIM) that directly embed user and item ID into a latent space and model user-item interaction via inner product.

Matrix Factorization

Matrix Factorization (MF) [27] is a classical method to perform collaborative filtering, that leverages the mathematical concept of matrix factorization to discover latent features for both users and items, and use those features to reconstruct the interaction matrix.

Given U, a set of users and I a set of items, with their respective interaction matrix $R \in \mathbb{R}^{|U|\times|I|}$, where some entries can be unknown (e.g. a user didn't interact with an item, so the rating is not known) and a hyperparameter k, indicating the dimensionality of latent features, the goal of matrix factorization is to reconstruct:

$$\hat{R} = P \times Q^\top$$

where $P \in \mathbb{R}^{|U| \times k}$ and $Q \in \mathbb{R}^{|I| \times k}$ are latent representations for users and items respectively. The formula can be also written in a non-vectorized way to highlight the contributions by P and Q:

$$\hat{R}_{ui} = Q_i^\top P_u$$

The training objective becomes a minimization of reconstruction mean squared error between $\hat{R}$ and R:

$$\min \sum_{(u,i) | R_{ui} \neq 0} (R_{ui} - \hat{R}_{ui})^2$$

where different variations of the model can include different regularization terms to be applied to the factorized matrices.

The matrix factorization approach can be also described as a deep learning problem in modern frameworks by simply embedding user and item ids to a latent space via embedding layers, and applying an inner product to obtain the reconstructed interaction matrix.

Factorization Machines

Factorization Machines (FM) [42] is a model that combines Support Vector Machines (SVM) with factorization techniques. They were proposed during the days when SVM were a wide-spread technique to be applied on Machine Learning problems. The main limitation of SVM was that they were too expensive to be applied on data with huge sparsity, such as recommender systems datasets.

Given $x \in \mathbb{R}^n$, an input vector composed of 4 subvectors: one-hot user id, one-hot item id, historical ratings and last rated item one-hot, the equation for factorization machine model is defined as:

$$\hat{y}(x) = \mathbf{w}_0 + \sum_{i=1}^{n} \mathbf{w}_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} (\mathbf{v}_i^\top \mathbf{v}_j) x_i x_j$$

where learnable model parameters are $\mathbf{w} \in \mathbb{R}^{n+1}$ and $\mathbf{V} \in \mathbb{R}^{n \times k}$. The described formula for the model is said to be a 2-way factorization machine (or FM of degree 2), because it captures all single and pairwise interactions between variables. Weights $\mathbf{w}_i$ are used to model the strength of i-th variable. Dot product $\mathbf{v}_i^\top \mathbf{v}_j$ is used to model interaction between i-th and j-th variable by means of factorization. Factorization allows the number of parameters to be n instead of n^2 when modeling variable interactions weights (i.e. we don't need to define a grid of weights $w \in R^{n \times n}$). The formula for d-way factorization machines can be found in the original work under section III.D .

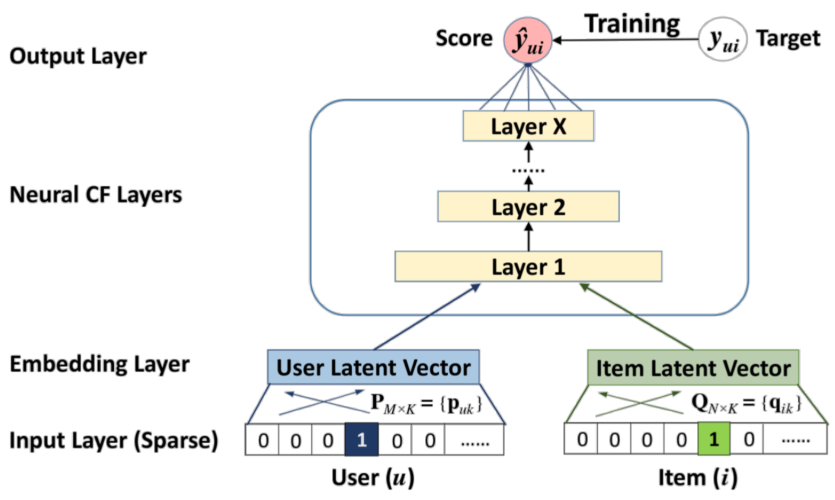
It is also worth noticing that thanks to the interaction factorization, the factorization machines are linear in complexity, allowing them to easily scale on very large datasets.

Sparse Linear Models

Sparse Linear Methods (SLIM) [39] is another work that proposed a modification of linear models to work on sparse learnable coefficients. The simplest model of such type can be defined as a sparse aggregation of items purchased by a user u_i:

$$\hat{a}_{ij} = a_i^\top w_j$$

which allows us to estimate the recommendation score a_{ij} of an user u_i and an unconsumed item t_j. The equation itself uses w_j, a sparse column vector of aggregation coefficients, which is a part of $W \in \mathbb{R}^{n \times n}$ a sparse matrix of aggregation coefficients. It is worth noting that given $a \in R^n$, a binary consumption vector for the user u_i, the term $\hat{a}_i = a_i^\top W$ of the previous equation represents the recommendation scores over all items consumed by u_i. Sorting



Source: "Neural Collaborative Filtering"

Figure 2.1: Architecture of Neural Collaborative Filtering model.

$\hat{a}_i$ in decreasing order, allows us to perform a top-N items recommendation. For details on how to train the model, please refer to IV.B of [39] .

—

Due to their simplicity, classical approaches have the limitation of only being able to capture simple and linear interactions. Recent, deep learning based approaches use more powerful interaction functions that can learn more complex and nonlinear relationships between users and items to overcome this problem. We will see two representatives of such models: Neural Collaborative Filtering and VAE-CF.

Neural Collaborative Filtering

Neural Collaborative Filtering (NCF or NeuroCF) [21] can be seen as a deep generalization of the Matrix Factorization approach illustrated before. The main concept that NCF extends is the interaction function, which the authors replace with an arbitrarily deep MLP. Using the same notation as in Matrix Factorization, we can define the equation of NCF as:

$$\hat{R}_{ui} = \text{MLP}(Q_i, P_u)$$

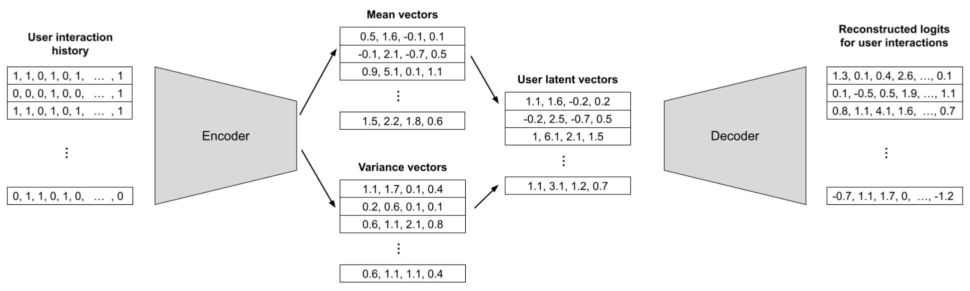


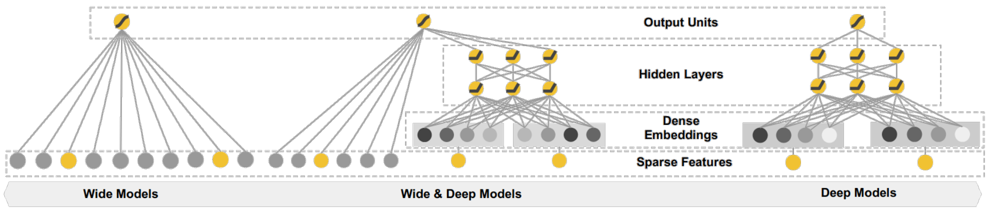
Figure 2.2: Architecture of VAE-CF model.

where Q_i and P_u are embeddings of items and users respectively, that can be obtained by the means of a learnable look-up table (i.e. embedding layers in modern deep learning frameworks).

Such extension is trained using reconstruction error and can yield to more powerful interactions being learned by the deep network. A visualization of the architecture can be seen at Figure 2.1.

VAE-CF

Variational Autoencoder for Collaborative Filtering (VAE-CF) [31] uses a Variational Autoencoder to reconstruct interaction vectors (items they have consumed) for users. Following VAE [25], the model receives in input a matrix of user-item interactions (submatrix in case of mini-batching). Encoder module is used to encode the interactions matrix into mean and variance vectors for each user, from which their latent representation will be sampled. Decoder module is used to decode the latent representation into a distribution over items that the user has interacted with. As in previous approaches, this distribution can be used to train with known ground truth entries of the user's interaction history. A visualization of the architecture can be seen in Figure 2.2.



Source: “Wide & Deep Learning for Recommender Systems”

Figure 2.3: Combination of wide models (left), expressed as generalized linear models, that work as a memorization module for numerical features, to capture interactions between them, and deep models (right) that process categorical features in a similar fashion as NCF does. Wide & Deep (center) merges the two types of models into a single one.

2.2.3 Hybrid Approaches

Hybrid Approaches aim at solving the drawbacks of both content-based and collaborative filtering models (e.g. the cold-start problem which will be described later in this chapter) by combining the two types of approaches together. Namely, any model that mixes both ids and content features can be viewed as hybrid. As those models use both user/item ids and user/item features the function they are trying to model becomes:

$$f(\text{user\_id, user\_features, item\_id, item\_features})$$

In this section we will describe some modern, deep learning based hybrid models: Wide-n-Deep, DeepFM and DLRM.

Wide-n-Deep

Wide-n-Deep [9] can be viewed as a neural generalization of the factorization machine. It is composed of two types of networks: wide and deep.

The wide component of the model is a generalized linear model that models the function:

$$y = \mathbf{w}^\top x + b$$

where $x = [x_1, ..., x_d]$ is a vector of d numerical features that can be both raw and transformed features. A possible transformation of features can be defined for categorical features (e.g. language, gender), called cross-product transformation:

$$\phi_k(x) = \prod_{i=1}^{d} x_i^{c_{ki}}$$

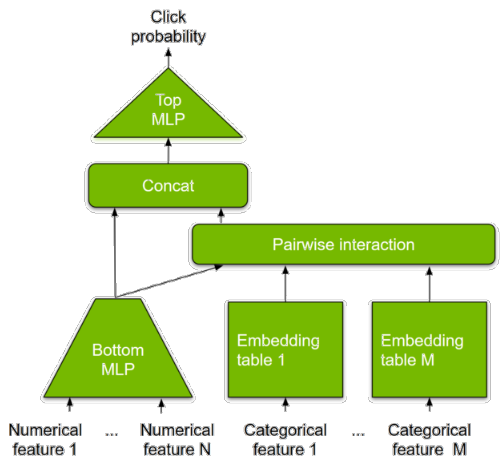
where $c_{ki} \in \{0, 1\}$ is a boolean variable that indicates whether i-th feature is a part of k-th transoformation ϕ_k. Such transformation captures interaction between categorical features and adds non-linearity to the linear model.

The deep component of the model can be seen as an extension of NCF architecture that supports any kind of categorical variables, not only ids, as in NCF case. It is used to model the interaction between categorical variables that would otherwise be too expensive to perform on the wide part. The training is performed jointly on wide and deep parts by using two different optimization algorithms (FTRL on wide part, AdaGrad on deep part) and formulating it as a logistic regression problem (predicting a click event). A visual summary of the architecture, as well as the models it's composed of can be seen in Figure 2.3.

DeepFM

DeepFM [17], similarly to Wide and Deep, combines factorization machines and deep neural networks. Similarly to Wide-n-Deep, it uses a deep neural network to model high-order feature interactions and a factorization machine for lower order ones (as opposed to generalized linear model from Wide-n-Deep). The key difference between the two however is that DeepFM does not use any manual feature engineering such as the cross-product transformations. It instead purely relies on the learning procedure to learn the appropriate features.

The factorization machine component, which as previously explained in Section 2.2.2, is used to capture pairwise (2nd order) feature interactions as



Source: “https://developer.nvidia.com/blog/optimizing-dlrm-on-nvidia-gpus/”

Figure 2.4: Architecture of Deep Learning Recommendation Model (DLRM).

inner product of feature vectors. Additionally, $w^\top x$ is used to capture the first order feature interactions (importance of features themselves).

The deep part on the other hand is used to learn generic, higher-order feature interactions in a deep learning fashion, without complicating or hard-coding the interaction order dimensionality in the original, factorization machine formula.

DLRM

Deep Learning Recommendation Model (DLRM) [38] is a giant, production-scale ready recommender by Facebook, that combines principles from both collaborative filtering and predictive analytics. Similarly to previously described models, DLRM processes its inputs by using two different components: embedding tables and MLPs.

Embedding tables are used in DLRM to encode all the categorical features to dense representations. Multi-layer perceptrons on the other hand are used to transform numerical features by means of a learnable component. A pairwise interaction operation (e.g. outer product) is used to compute second order interactions between both categorical features and processed numerical features. Jointly, the computed interactions are concatenated with processed numerical features and passed to another MLP whose goal is to predict a probability of consumption. A summary of the architecture can be seen in Figure 2.4.

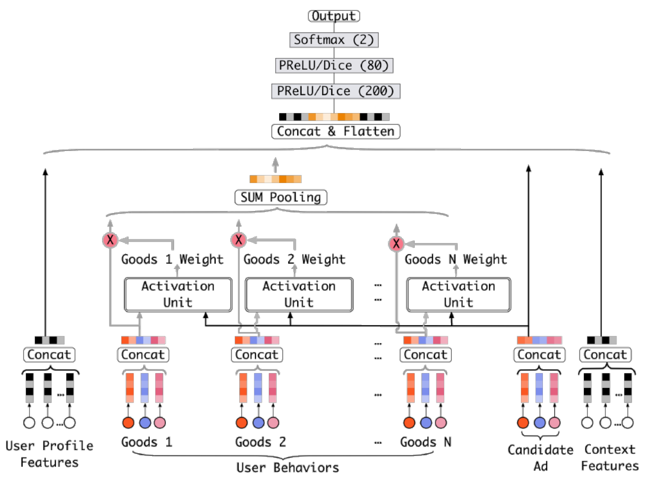
Despite the simplicity of the architecture, it can contain up to tens of trillions of parameters [37], magnutude of orders more than the famous GPT-3 175 billion parameter model for deep language generation. This required authors to introduce a hybrid-parallelism schema, that would allow the model to run in a model-parallel fashion (running parts of the same model over multiple workers) for the embedding tables, and in a data-parallel fashion for the MLPs (running the same version of the model on different data).

2.2.4 Sequential

The previously defined approaches generally ignore temporal dynamics and sequences of interactions when modeling user behavior. Sequential recommenders formulate the recommendation problem as a task of predicting consumption of an item, given a history of previously consumed items. For example, given a sequence of played songs, what song the user might like to play next, or given a sequence of purchases, which item will the user be interested in purchasing. Generally, they model the function:

$$\mathbb{P}(\text{target}|\text{sequence, features})$$

Former, classical approaches, based on Markov Decision Process (MDP) would treat recommendation generation based on previous items as a sequential optimization problem [45]. Recent methods based on deep learning were initially using recurrent neural networks. The key idea in deep learning based methods is to embed the whole user behaviour sequence into a fixed size representation, which combined with other features such as user profile features, target item features would help predicting the probability of consumption. We will go over three sequential models: DIN, DIEN and SIM.



Source: "Deep Interest Network for Click-Through Rate Prediction"

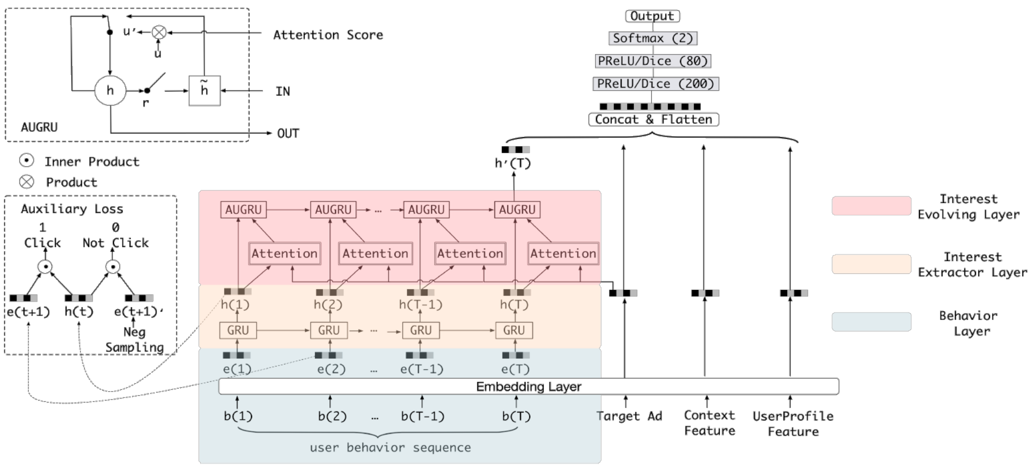
Figure 2.5: Deep Interest Network architecture first embeds all the given user, item sequence, target item and context features via appropriate embedding tables. It then uses an attention mechanism to produce scores between target item and historical items, and uses those scores as coefficients for a sum of historical items, producing a single representation from the full sequence. All the embeddings are then concatenated and passed to a classification head to predict likelihood of consumption.

DIN

Deep Interest Network (DIN) [74] is a simple deep learning based sequential recommender that works by performing a weighted sum pooling on a given, fixed length, sequence of user's consumption history. The weight coefficients for the sum are estimated by using an attention mechanism between each item in the sequence and a given target item. Lastly the pooled representation, together with user's features and target's features are passed to a classification head that outputs a probability of consumption. A schematic overview of DIN's architecture is provided in Figure 2.5.

DIEN

Deep Interest Evolution Network (DIEN) [73] is an extension of the DIN model that instead of a simple weighted sum pooling, uses a recurrent neural network to produce a summary representation of the historical sequence. The recurrent network is based on GRU and AUGRU layers, where the latter



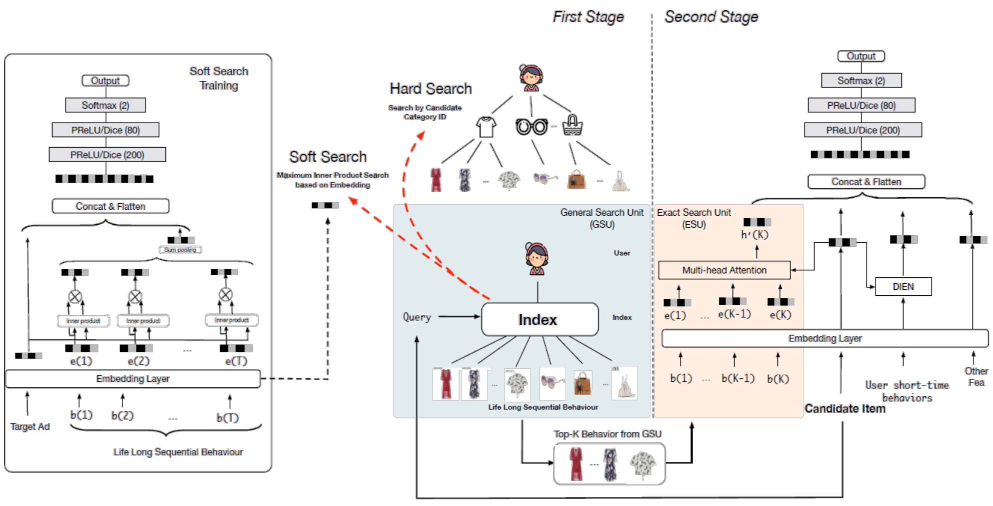
Source: "Deep Interest Evolution Network for Click-Through Rate Prediction"

Figure 2.6: DIEN architecture is an extension of DIN, where a stack of recurrent cells is added to handle historical sequences of variable lengths. As in DIN, an attention mechanism between target embeddings and intermediate representations of individual items in the sequence is used to better model the whole sequence embedding. Intermediate sequential item representations are further refined via an auxiliary loss with negative sampling at each time step, to learn sequentiality patterns between items at consequent time-steps in a self-supervised way.

are a modification of GRU that includes an attention mechanism. Additionally, the training uses an auxiliary loss applied to the intermediate recurrent network's representations at each time step of the sequence. This auxiliary loss acts as a self-supervised schema to learn the sequentiality between the items in the sequence, as it is applied to consequent steps. A more detailed, visual description of the architecture is provided in Figure 2.6.

SIM

Search-based user Interest Modeling (SIM) [40] is a further extension over DIEN model that uses a two stage architecture to overcome memory related problems that arise when feeding large sequences to the recurrent layers. Indeed, DIN used a simple sum pooling to encode the sequence, which didn't require any particular calculation aside from attention to be performed. DIEN's recurrent network on the other hand, requires keeping the internals of all the



Source: "Search-based User Interest Modeling with Lifelong Sequential Behavior Data for Click-Through Rate Prediction"

Figure 2.7: SIM architecture. Here the first stage is depicted with Hard and Soft search.

recurrent layers to both perform the backward pass, drastically increasing the memory usage. Thus, to make an efficient usage of the two previous models, SIM splits the user's historical sequence into "long" and "short" subsequences and processes them separately in two stages. More specifically, a first stage called General Search Unit (GSU) uses a DIN model to select a set of top-K items similar to target item, from a long historical sequence of items. The top-K most similar items are then passed to a second, more computationally expensive stage, called Exact Search Unit (ESU). The Exact Search Unit first builds a single representation out of the top-K items by using a multi-head attention mechanism between target item's representation and the top-K items. Then, the relatively expensive DIEN block is used to build a representation for short-term historical behaviors. Finally, all the produced representations (pooled top-K, pooled short, target and user's features) are concatenated and passed to a classification head as in previous models. A visual summary of the architecture is provided in Figure 2.7.

2.3 Problems

There are open problems in the recommendation field that might apply to the previously described architectures.

2.3.1 Cold start

Cold start is a problem that occurs when the system needs to provide inferences about users, items or communities for which insufficient information has been gathered (e.g. users who have rated only a few items, items that were not bought by anyone yet), making it hard for the system to output meaningful recommendations. One way of mitigating this problem is to use hybrid recommendation, which would use collaborative filtering for “warm” predictions (the ones that have enough interactions) and content-based filtering for “cold” ones (that do not have enough of them yet). This is possible because content-based filtering does not require any prior interaction information to recommend items.

2.3.2 Data sparsity

Data sparsity problem arises from the fact that users generally interact with a smaller fraction of all the available items, meaning that the system would have insufficient interaction data to cluster, compromising the overall quality of the recommendations [46].

2.3.3 Embedding table sizes

Collaborative filtering recommender systems are characterized by having huge embedding tables, used to encode categorical variables’ values (e.g. user ids, item ids) to the latent space. For example, DLRM model implementation by NVIDIA[2] has checkpoints that can occupy around 140GB most of which are

[2]`https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Recommendation/DLRM`

due to embedding tables.

2.3.4 Overspecialization

Overspecialization is a problem that occurs when users are prevented from discovering new items due to the recommendations resembling those already known to the user or even to those defined in their profiles (e.g. a user has brought a smartphone and gets recommended the same smartphone, or another one which the user was already familiar with). Approaches that deal with this problem might be considered a bit risky, as they try to sacrifice the recommendation relevance in favor of fresh discoveries [1].

Chapter 3

Graph Neural Networks

Graph Neural Networks (GNNs) are a family of neural networks that work on inputs organized in a graph structure (e.g. social networks, road traffic maps, molecule graphs). Their goal is to learn a suitable representation of the input graph data, in terms of node features, while having access to the connectivity structure of the graph (i.e. neighbors information). The learned node representations are then usually passed to an appropriate task head for the problem we're trying to solve.

The need of a new family on neural networks arises from the primary challenge we might encounter when trying to apply deep learning methods on graph-structured data, that is, typical approaches are defined for precise input structures. For example, recurrent neural networks (RNNs) assume working with data organized in sequences (e.g. text, temporal phenomenon) and convolutional neural networks (CNNs) assume working with grid-structured data (e.g. images, videos). Both mentioned data types could be seen as a particular case of graph structure. However, in order for deep neural networks to work on generalized graph structures we need to use different types of deep learning architectures.

In the following sections we are going to see the main concepts that drive the properties and architectures of graph neural networks, the commonly used types of graph neural network modules, solving different problems on graphs,

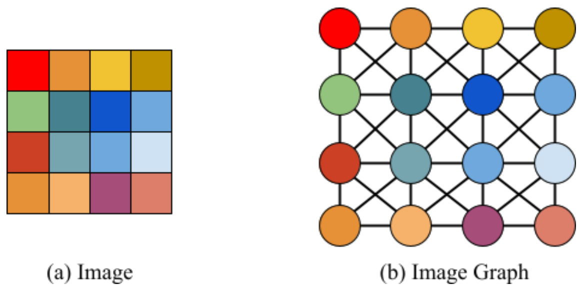


Figure 3.1: An image (a) can be represented by a graph (b) with a connected neighborhood of at most 8 nodes, where each node has RGB features associated to it.

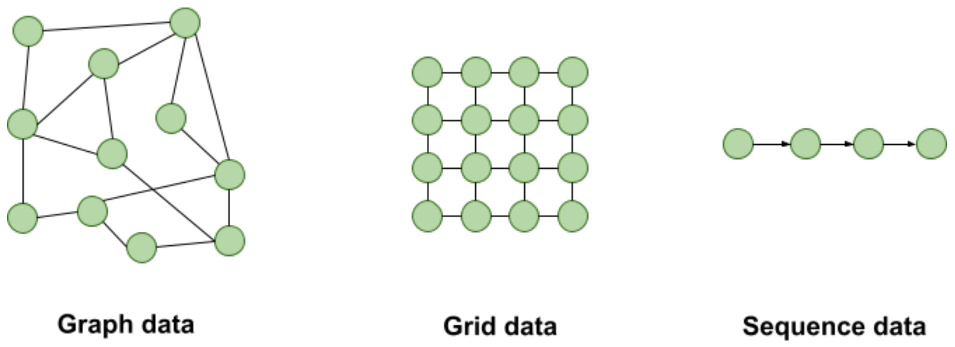


Figure 3.2: Different types of commonly used data. Graph data can be seen as a generalization over both grid and sequence data, where the prior structure can be arbitrarily defined.

ways to improve the scalability on bigger sized datasets.

3.1 Main concepts

In the following section we are going to describe the main concepts that enable graph neural networks: the nature of non-Euclidean data, permutation equivariance and invariance, graph neighborhoods, concepts of neural message passing.

3.1.1 Non-Euclidean space data

Majority of the existing deep learning approaches are built to be used with Euclidean or grid-like structures, such as images, videos or textual data. Images for example can be seen as a function on the Euclidean space (plane), sampled on a grid, allowing us to exploit their local connectivity and to use Convolutional Neural Networks that exploit this prior about the data. Textual data can be also represented as a sequence on a Euclidean plane, on which we also have structural notions of "before" and "after" when representing the text that NLP models exploit.

On the other hand, data like social networks in computational social sciences, sensor networks in communications, molecule structure in computational chemistry, meshed surfaces in computer graphics and many more, can be all seen as examples of non-Euclidean space data. The non-Euclidean nature here generally means that there are no common systems of coordinates, data priors or common structures that represent such data [6]. Therefore, basic approaches that work on Euclidean data, fail to work on its generalization, non-Euclidean case, where prior structure can be arbitrarily represented (see Figure 3.2).

It is also worth noting how the Euclidean data can be seen as a particular case of non-Euclidean data. For example, an image grid of $N \times N$ pixels can be viewed as a graph with N^2 nodes and at most 8 edges per node (connecting to the nearest grid of pixels) with each node associated a feature vector representing the image's pixel intensity, as seen in Figure 3.1.

3.1.2 Permutation equivariance and invariance

When applying deep learning methods to graph data, a common and reasonable idea that we can think of would be that of applying a neural network, such as a multi-layer perceptron (MLP) to the adjacency matrix of the graph as input. With such approach, we could attempt to generate graph embeddings $\mathbf{e}_{\mathcal{G}}$ from a concatenation of rows of a flattened adjacency matrix $\mathbf{A}$:

$$\mathbf{e}_{\mathcal{G}} = \text{MLP}([\mathbf{A}[1] \oplus \mathbf{A}[2]\mathbf{A} \oplus ... \oplus \mathbf{A}[|V|]])$$

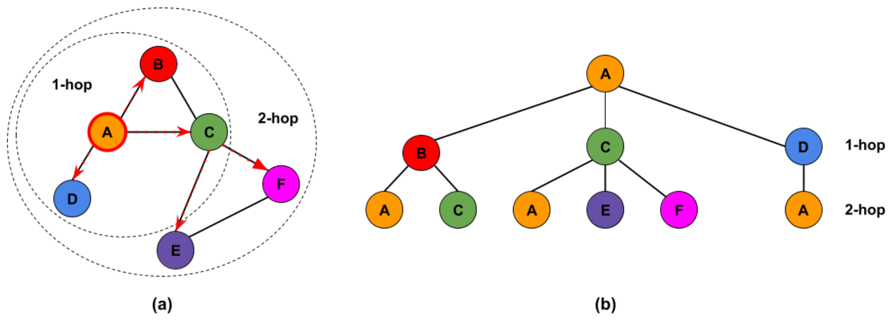


Figure 3.3: (a) 1-hop and 2-hop neighborhoods of a given target node A. (b) Tree structure corresponding to the 2-hop neighborhood of node A. Tree structures of neighborhoods implicitly represent computational graphs that can be exploited when designing different types of GNN layers.

This approach would be dependent on the order of nodes that was used in the adjacency matrix. In other words, if we had to feed the same graph structure but with nodes passed in a different order, then the embedding of the graph would be different. In fact, such an approach is said to be not *permutation invariant*.

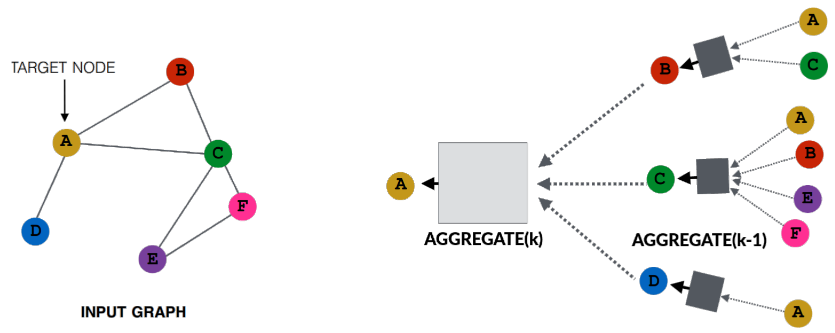
The key property to design neural networks that work over graphs is that they should be *permutation invariant* or *equivariant*. More formally, a function f that operates on an adjacency matrix $\mathbf{A}$ as input, should satisfy one of two following properties:

$$f(\mathbf{PAP}^\top) = f(\mathbf{A}) \qquad \text{(Permutation Invariance)}$$
$$f(\mathbf{PAP}^\top) = \mathbf{P}f(\mathbf{A}) \quad \text{(Permutation Equivariance)}$$

where $\mathbf{P}$ is a permutation matrix. Intuitively, permutation invariance means that the function f does not depend on the ordering of rows/columns of the adjacency matrix. Permutation equivariance, similarly, means that the output of f is permuted in a consistent way when the adjacency matrix is permuted.

3.1.3 Graph neighborhood

Given a graph $\mathcal{G}$, the neighborhood of a given node u is the set containing all the nodes of the graph that are adjacent to u, denoted in this work as $\mathcal{N}(u)$



Source: "Jure Leskovec, Stanford CS224W: Machine Learning with Graphs, http://cs224w.stanford.edu"

Figure 3.4: For a given input graph and a target node A, the new feature representations can be computed by taking representations from A's neighbors' {B, C, D} aggregated representations, whose features in their turn are computed by taking their neighbors' representations. This visualization represents two layers of a message passing model and can be represented using a tree structure by unfolding neighborhoods around target the node.

or $\mathcal{N}_u$. Despite $\mathcal{N}(u)$ being a set of direct neighbors of u, some authors also include u itself into the neighborhood set. An extension of this concept is the k-hop neighborhood, which denotes a set that contains all the nodes distant at most k connections from a given node u, as visible in Figure 3.3 (a).

Another property of neighborhoods is that they implicitly define tree structures with root being the given node, as can be seen in Figure 3.3 (b). This fact will be exploited in further sections when defining message passing, as well as different types of GNN layers.

3.1.4 Neural message passing

When working with graphs, we are more specifically dealing with information that is associated to their nodes, in terms of node features, and to the graph connectivity, in terms of edges, as defined in Section 3.1.1. In order for graph neural networks models to learn effective node features/representations, we need to introduce the concept of *neural message passing*.

At its core, neural message passing assumes that each node $u \in \mathcal{V}$ in a given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, is associated a feature vector $\mathbf{x}_u \in \mathbb{R}^d$, where d is some feature dimension. In order to update a node u's feature vector, producing a new feature vector $\mathbf{h}_u^{(k)}$, we need to be able to collect feature information coming from the node's neighbors, as well as to integrate this information, to produce a new representation for the node. In terms of message passing, we need two functions AGGREGATE and UPDATE that can be used in the following way to produce a new representation:

$$\begin{aligned}\mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right)\\ \mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right)\end{aligned}$$

where AGGREGATE and UPDATE are both arbitrary differentiable functions (e.g. neural networks) and $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$ is said to be a "message" that is aggregated from u's neighborhood $\mathcal{N}(u)$.

Note: since the AGGREGATE operation operates on sets as inputs, GNNs defined in this way are permutation equivariant by design.

The superscript (k) is used to indicate embeddings and the functions at different layers/iterations of message passing. In fact, at each layer/iteration k of this procedure, the AGGREGATE function takes in input a set of embeddings of node u's neighbors $\mathcal{N}(u)$, $\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\}$, and constructs a message $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$ based on the aggregated neighbor information. The UPDATE function then combines this message $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$ with the previous node u's embedding $\mathbf{h}_u^{(k-1)}$ to generate an updated embedding $\mathbf{h}_u^{(k)}$.

After performing K iterations of message passing, we can use the final representation at step K to define embeddings for each node as:

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}$$

Final representation of the nodes computed in the previously described way allows the model to encode both feature-based and structural information.

Structural information is the information that implicitly comes from graph connectivity, such as degrees of all the nodes in a k-hop neighborhood, which finds usages in fields like molecular property prediction or recommender systems, to use implicit graph structure that other, non-graph models would ignore or have to explicitly make use of.

Feature-based information is the information captured from the actual node embeddings. Thanks to the way message passing is defined, nodes aggregate features incoming from their neighbors, allowing them to encode features from the entire k-hop neighborhood, similar to what Convolutional Neural Networks do with local spatial information.

As stated before, since GNNs perform local feature aggregation over the graph structure, the number of layers in a GNN is a hyperparameter that has effect on how many neighborhood hops the network will perform from a single central node. Setting this hyperparameter to a high value (i.e. more than 6) can actually result in over smoothing of features, due to too many layers being stacked.

3.2 Types of GNNs

Having described some of the theoretical motivations behind graph neural networks in the previous section, including an abstract definition of neural message passing, in this section we will describe how different types of concrete GNN layers can be defined in terms of both neighborhood aggregation and update functions.

When talking about different types of GNN layers, what mostly changes between the variants is the way UPDATE and AGGREGATE functions are defined. Sometimes also the choice of neighbor sampling type or the choice of non-linearity operations affect the layer. Typical examples for update functions include mean, max, neural network and recurrent neural network, to be applied to the incoming message and to previous node's state. For aggregation functions there is typically mean pooling, max pooling, normalized sum pooling and neural network, applied to all the incoming information from the neighbor nodes.

An early work, for example, Graph Convolutional Networks [26] (described later) uses a normalized sum of neighbor embeddings as aggregation function and incorporates update function inside aggregation by adding a self-loop. Graph Attention Networks [57] work uses attention weights inside the aggregation function to weight the sum based on an attention score associated to each pair of nodes.

3.2.1 Simple Neighborhood Aggregation

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with N nodes ($|\mathcal{V}| = N$) and its adjacency matrix $A \in \mathbb{B}^{N \times N}$, the simplest kind of a GNN layer that uses neighborhood aggregation to aggregate features associated with a node i could be defined as:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \mathbf{W} h_j^{(l)}\right) \tag{3.1}$$

Here we assume that $\mathbf{h}_i \in \mathbb{R}^K$ is a feature vector associated with i-th graph node and $\mathbf{W} \in \mathbb{R}^{K \times D}$ is a learnable transformation matrix that maps feature dimensions from K to D, producing a new node representation $\mathbf{h}_i^* \in \mathbb{R}^D$.

Noting that an adjacency matrix A of a graph represents neighborhood situation for each node, we could define $\tilde{A} = A + I_N$ to extend the adjacency matrix with self-loops and rewrite (3.1) to a more compact and efficiently computable vector form:

$$H^{(l+1)} = \sigma(\tilde{A}\mathbf{W}H^{(l)}) \tag{3.2}$$

3.2.2 Graph Convolutional Networks

Graph Convolution (GCN) [26] extends the neighborhood aggregation concept described in the previous section by incorporating a normalization factor, which is computed based on nodes' degrees. In particular it uses a normalization matrix $S = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$, where $\tilde{D}$ is the degree matrix with diagonal

$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. The role of this new degree matrix is to weight the incoming embeddings of neighbors v_j by the factor $\frac{1}{\sqrt{deg(v_i)deg(v_j)}}$, when performing the aggregation process for node v_i. We can write the formula for computing aggregation of a GCN, based on 3.1:

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}_i} \frac{1}{S_{ij}} \mathbf{W}^{(l)} h_j^{(l)}\right)$$

The normalization induced by S is called symmetric normalization in the GNN literature. There exists also "left" normalization, that corresponds to normalization of messages by each node's in-degrees, which is equivalent to averaging the received messages: $\frac{1}{deg(v_i)}$.

3.2.3 Graph Attention Networks

Graph Attention Networks (GAT) [57] are another generalization over the graph convolutional networks, where the influence of different neighboring nodes is weighted by a dynamically calculated attention score, rather than with a constant factor during the aggregation operation. The feature update formula can be described by the following equation:

$$\begin{aligned}
\mathbf{z}_i^{(l)} &= \mathbf{W}^{(l)}\mathbf{h}_i^{(l)} \\
e_{ij}^{(l)} &= \text{LReLU}(a^{(l)}(\mathbf{z}_i^{(l)}||\mathbf{z}_j^{(l)})) \\
\alpha_{ij}^{(l)} &= \frac{\exp(e_{ij}^{(l)})}{\sum_{k\in\mathcal{N}_i}\exp(e_{ik}^{(l)})} \\
\mathbf{h}_i^{(l+1)} &= \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^{(l)}\mathbf{z}_j^{(l)}\right)
\end{aligned}$$

where initially the feature vector $\mathbf{h}_i^{(l)} \in \mathbb{R}^D$ is mapped to a D' dimensional vector via the transformation $\mathbf{W}^{(l)}$. Then, a function $a^{(l)} : \mathbb{R}^{D'} \times \mathbb{R}^{D'} \to \mathbb{R}$, called "attention module", is applied pairwise on the central node and its neighbors to estimate an attention coefficient. This function $a^{(l)}$ can be an arbitrarily complex function such as a neural network. Finally, a distribution of attention coefficients over all the neighbors of the node is computed and used as a normalization score in the aggregation.

3.2.4 GraphSage

In case of graph neural networks the existing approaches can be classified into inductive and transductive approaches:

- **Transductive**: all the nodes evaluated at the testing time were observed also at the training time.
- **Inductive**: nodes present at testing time were not necessarily present in the training data.

GraphSAGE [18] work can be seen as an inductive extension over the Graph Convolutional Networks and Graph Attention Networks. The idea of their approach is to consider only a fixed size neighborhood by sampling the full neighborhood of a given node when performing aggregation step for it, which allows to both make the aggregation step more efficient as well as to work on nodes unseen in the training set as the full graph Laplacian matrix is no longer required to perform aggregation.

Additionally to this inductive extension, the work also introduces a method to learn node representations in an unsupervised setting by defining the following loss function:

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n}))$$

where v is a node that co-occurs near u on a fixed-length random walk and P_n a negative sampling distribution with Q negative samples. Such loss function encourages nearby nodes to have similar representations while enforcing highly distinct representations for more distant nodes.

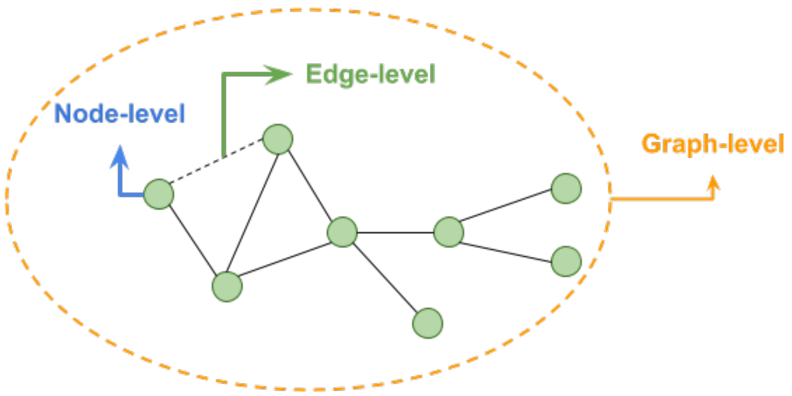


Figure 3.5: Various common levels of tasks solvable on graph structures by machine learning approaches that work on node embeddings. Node-level tasks consist in predicting the property of an individual node (e.g. age of an user in a social network graph). Edge-level tasks consist in predicting property or an existence of an edge (e.g. rating that someone gave to a product, whether a transaction happened or not). Graph-level tasks consist in predicting a property of an entire graph (e.g. type of a molecule given a molecule graph).

3.3 Problems on graphs

Having described how basic types of layers allow to compute new representations of node embeddings when exploiting the graph structure, this section will briefly describe general techniques that can be applied to the node embeddings to solve different tasks at node, edge and graph level, as shown in Figure 3.5.

3.3.1 Node-level tasks

A standard way to tackle a node-level problem when using a graph neural network is to simply apply a task head (e.g. classification head) on top of the learned node features, for each node we're interested in. The motivation behind this is that after applying graph layers, the representations generated by them generally do not need to keep the information about graph structure anymore, and can be treated as simple vector data processable by any classical machine-learning or deep learning model that works on vectors. Below is an example of a linear classifier applied to the embeddings of nodes, $\{e_i | i \in$

$\mathcal{V}_{train}\}$ learned by a GNN:

$$\mathbf{s}_i = \mathbf{W}^{(clf)} e_i + \mathbf{b}^{(clf)}$$
$$\mathcal{L} = \sum_{i \in \mathcal{V}_{\text{train}}} -\log(\text{softmax}(\mathbf{s}_i))$$

3.3.2 Edge classification

Edge-level tasks on the other hand require to first compute a pairwise interaction function between node embeddings, involved in the edges, to generate an edge embedding:

$$\mathbf{e}_{uv} = \text{interaction}(\mathbf{h}_u, \mathbf{h}_v)$$

Interaction functions generally used in the literature are dot product (especially in recommendation tasks), concatenation and MLP:

$$\mathbf{e}_{uv} = \mathbf{h}_u^\top \mathbf{h}_v \qquad \text{(Dot product)}$$
$$\mathbf{e}_{uv} = \mathbf{h}_u \parallel \mathbf{h}_v \qquad \text{(Concatenation)}$$
$$\mathbf{e}_{uv} = \text{MLP}(\mathbf{h}_u \parallel \mathbf{h}_v) \qquad \text{(MLP)}$$

The embedding $\mathbf{e}_{uv}$ can then be used to perform tasks at edge-level, such as edge classification, by applying the appropriate task head to it.

In Chapter 4, we will see more examples of how edge classification is performed when dealing with recommender systems based on graph neural networks.

3.3.3 Graph-level tasks

Finally, graph-level tasks generally require embeddings to from all the nodes of the graph to make a prediction [69][7][4]. In order to combine node features across the whole graph, a **readout** operation is used:

$$\hat{\mathbf{y}} = \text{READOUT}(\{\mathbf{h}_v | v \in \mathcal{V}\})$$

where each $\mathbf{h}_v$ is assumed to be the representation of node v at last graph layer and READOUT can be an arbitrarily complex function, such as a neural network or as simple as a mean, defined below:

$$\hat{\mathbf{y}} = \frac{1}{|\mathcal{V}|} \sum_{v \in \mathcal{V}} \mathbf{h}_v^{(K)}$$

As for previous tasks, the final embedding $\hat{\mathbf{y}}$ can be used further by a specialized task head, trained appropriately.

3.4 Scaling to larger graphs

Real world graphs are typically large in size, both in terms of nodes and edges. For example, typical recommendation systems are composed by 100M to 1B users and 10M to 1B products; social networks too can include from 300M to 3B users; Microsoft academic graph [47] consists of 120M nodes representing authors and papers. Graphs of such sizes are too large to fit on a GPU due to memory limitations, even on higher end GPUs[1]. General techniques such as mini-batching that work on typical deep learning models, cannot be naively applied to GNNs due to the inherent structure of both graphs and GNN layers. Graph layers work by aggregating features over their neighbors to produce better representations, thus when sampling naively nodes from a graph, there is no guarantee that the sampled subgraphs will be connected at all, preventing us from effectively training the model.

This section will describe some techniques that can be used to increase model performance and to allow training them on larger graphs. The described techniques can be divided into two families for which we will report only a couple of works:

- **Subgraph message passing**: performing message passing over smaller subgraphs of the original graph.

[1]At the time of writing, NVIDIA's A100 80GB GPU is considered to be the largest GPU available on the market

- **Model simplification**: simplifying the feature preprocessing operation performed by the graph model.

3.4.1 Subgraph message passing

Subgraph message passing techniques are based on performing message passing over smaller subgraphs of the original large graph. These techniques mainly differ by the way they construct the subgraph.

GraphSAGE

GraphSAGE, explained previously in 3.2.4, was a work that changed the way we train GNNs and how we create mini batches of a graph.

The main observation behind GraphSAGE is that in order to generate the representation for a central node, a K-layer GNN needs to consider only a K-hop neighborhood structure and its features, ignoring the rest of the graph. This observation can be used in generating mini-batches by considering them as sets of M different nodes. We can generate the embeddings for M nodes in the mini-batch by using M computational graphs (see tree structure in 3.3), which are more likely to fit on a GPU. In other words, using this paradigm, we can build mini-batches by first sampling M nodes and then placing their K-hop neighborhood computation graphs on a GPU, averaging the loss over M nodes in the training phase.

However, there's still a problem due to neighborhoods introducing an exponential amount of nodes by each new hop on the computational graph (i.e. number of nodes in computational graphs is exponential in its depth). This problem can be alleviated by rendering the computational graph more compact, using neighborhood sampling techniques.

We can construct the computational graph by sampling at most H_k neighbors at each hop k, where $k \in \{1..K\}$. This way, the K-th layer of the GNN will involve at most $\prod_{k=1}^{K} H_k$ leaf nodes in the computational graph. Indeed, while this would still result in an exponential computational graph, the growth can now be bounded by a set of hyperparameters $H = \{H_1, ..., H_K\}$, which in practice are set to have smaller fan-outs.

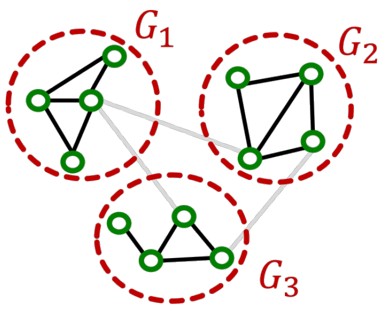
The introduction of H fan-outs hyperparameters introduces a trade-off between the amount of nodes to sample at each layer and the training stability, as using smaller H_ks would result in an unstable training due to high variance.

Lastly, the choice of sampling technique for sampling neighbor nodes is also important as not all the neighbors might have the same importance towards the central node's embeddings. Some of the typically used sampling techniques in the literature are:

- **Random sampling**: randomly sample neighbors of a node, by assigning a uniform probability to every neighbor to be sampled. This results in a fast but not very effective approach, as it might sample too many unimportant nodes.
- **Weighted random sampling**: randomly sample neighbors of a node, while assigning a weight to each neighbor when performing the sampling.
- **Random walk with restart**: sample neighbors of a node with highest "random walk with restart" score R_i, that indicates the probability of ending up in node i when starting from a central node. A random walk with restart is a normal random walk on a graph, that also includes a probability to restart the walk from the central node.

Clustering

As mentioned previously for GraphSAGE, the size of a computational graph becomes exponentially large with respect to the number of GNN layers we use. Additionally to the computational graph size, most of the computation is redundant in case nodes within the mini-batch have many neighbors in common. There are two ways of dealing with this situation: Hierarchical Aggregation



Source: “Jure Leskovec, Stanford CS224W: Machine Learning with Graphs, http://cs224w.stanford.edu”

Figure 3.6: Induced partitions $\mathcal{G}_1, ..., \mathcal{G}_3$ of a given graph. Note that the gray edges indicate edges that are excluded from the sampling of vanilla Cluster-GCN, as those are edges that connect different clusters. “Stochastic Multiple Partitions” on the other hand might include them when forming bigger partitions across multiple induced graphs.

Graphs [23] and Cluster-GCN [10]. We will only focus on the Cluster-GCN approach.

The motivation behind Cluster-GCN is that in full batch setting, layer-wise node embedding update allows to reuse embeddings from the previous layer, significantly reducing the computational redundancy introduced by neighbor sampling (i.e. aggregation is linear in the number of graph edges). Of course, layer-wise update on the full graph is still unfeasible on the larger graph due to limited amount of GPU memory as already described. However, we could sample a smaller subgraph of the former, large graph and perform efficient layer-wise node embeddings update over this smaller subgraph.

In order to be effective, the smaller subgraph should reflect the edge connectivity structure of the large graph as close as possible, allowing to generate embeddings similar to the ones in the original graph. As a matter of fact, real world graphs exhibit a community/clustering structure, allowing us to decompose a large graph into many smaller subgraphs. Those community structures can be sampled as subgraphs that would ideally retain essential local connectivity patterns of the original graph.

The vanilla Cluster-GCN approach consists of two steps:

1. **Preprocessing**: given a large graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, partition $\mathcal{V}$ into C

groups $\mathcal{V}_1, ..., \mathcal{V}_C$, with any community detection algorithms present in literature, such as Louvain Method [5] or METIS [24]. The partitioned node groups $\mathcal{V}_1, ..., \mathcal{V}_C$ induce C subgraphs $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1), ..., \mathcal{G}_C = (\mathcal{V}_C, \mathcal{E}_C)$ where $\mathcal{E}_c = \{(u, v)|u, v \in \mathcal{V}_C\}$ (i.e. induced subgraphs are disconnected between each other, as seen in Figure 3.6).

2. **Mini-batch training**: for each mini-batch, randomly sample a node group $\mathcal{V}_C$, construct the induced graph $\mathcal{G}_C = (\mathcal{V}_C, \mathcal{E}_C)$ and apply the GNN, layer-wise, on the whole graph $\mathcal{G}_C$ as usual.

There are however a couple of issues with the vanilla Cluster-GCN. First, the induced subgraph removes the between-groups links, which can affect the model's accuracy. Second, the sampled node group tends to only cover a small, concentrated portion of the entire data, making them not diverse enough to represent the entire graph structure, leading to unreliable gradients (high variance in gradients) and thus slower SGD convergence.

A solution to the vanilla Cluster-GCN problems is called by the authors as "Stochastic Multiple Partitions", which consists in aggregating multiple node groups into the individual induced subgraph (i.e. connect multiple induced subgraphs between themselves).

Summarizing the computational efficiency, GraphSAGE samples H nodes per layer, which for M nodes would result in $O(MH^K)$ cost in terms of both memory and compute time for the message passing. Cluster-GCN on the other hand requires $O(KMD_{avg})$ memory and compute time for the message passing, where D_{avg} is the average node degree. Cluster-GCN thus results to be linear with respect to K, unlike GraphSAGE which is exponential. In practice, however, as K tends to be small for real world graphs, GraphSAGE is preferred due to its flexibility.

3.4.2 Model simplification

Instead of focusing on the graph dimension, we can try to focus on model dimension and attempt to improve graph neural networks' performance by simplifying the model in terms of operations it uses. In this section we will describe the way Simple Graph Convolution [63] simplifies the graph convolution operation.

Recall that the original GCN formulation as described in 3.2.2 can be written as:

$$h_i^{(l+1)} = \text{ReLU}\left(\frac{1}{|\mathcal{N}_i|}\sum_{j\in\mathcal{N}_i} W^{(k)}h_j^{(k)}\right)$$

or more compactly in vector form as:

$$H^{(k=1)} = \text{ReLU}(\tilde{A}H^{(k)}W_k^\top)$$

where $\tilde{A} = D^{-1/2}AD^{-1/2}$ is a reformulation of the $\tilde{A} = D^{-1}A$ that works better empirically, and D^{-1} the inverse of degree matrix, such that:

$$D_{v,v}^{-1} = 1/|\mathcal{N}_v|$$

Now, if we remove the non-linearity from the above formula, then we end up with the following formulation:

$$H^{(k+1)} = \tilde{A}H^{(k)}W_k^\top$$

that we can unroll to obtain:

$$H^{(k+1)} = \tilde{A}H^{(k)}W_k^\top = ... = \tilde{A}^K XW^\top$$

where $W = W_{k-1}...W_0$ is a single linear weight matrix. $\tilde{A}^K = \tilde{A} \times ... \times \tilde{A}$ is a K times exponentiation of the adjacency matrix which represents a K-hop neighborhood connectivity with a given central node and X the initial node feature matrix. The key benefit here is that we can precompute $\tilde{X} = \tilde{A}^K X$ offline on a CPU and only use $\tilde{X}$ at the training time, transforming the graph layer into a simple linear projector: $H^{(K)} = \tilde{X} W^\top$.

As for the advantages of this approach, the computational cost of linearly transforming an individual node embedding, $h_v^{(k)} = W\tilde{X}_v$, for M nodes is linear in M. When compared to neighborhood sampling, this formulation does not require us to construct giant computational graphs. The advantage is also over ClusterGCN, as this way nodes can be sampled at random in the mini-batches, due to the aggregation step being done offline, before performing the actual training computation on GPU.

The main disadvantage, on the other hand, comes from the fact that a simpler model is less expressive, and thus would result in a worse accuracy when compared to more complex deep learning models. However, in real world, simplified GCNs tend to work well compared to original GCNs in semi-supervised classification benchmarks due to a phenomenon called graph homophily: nodes that are connected by edges end to share same target labels (e.g. two papers more likely share the same category if they city each other, two users tend to like same movie types if they are connected on a social network).

Chapter 4

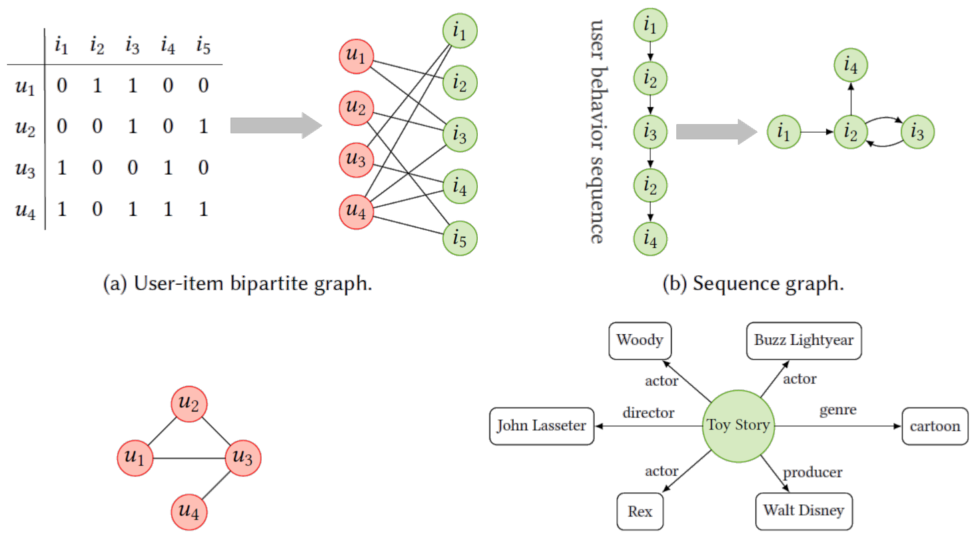
Graph Recommender Systems

In previous chapters we have seen an overview of Recommender Systems as a field and Graph Neural Networks as an approach to work on data structured via graphs. In this chapter we are first going to see how recommendation tasks can be formulated as problems on graphs and how different, graph based, architectures can be then leveraged to solve the problems in question. Lastly, we will characterize the models based on some structural properties, such as model size or the type of sampling, that can affect their performance properties. We will focus only on architectures that offer an official, working code implementation to both narrow down the scope of models, and to make sure we can extract some information relevant to the characterization.

4.1 Data Representation

When working with recommendation datasets, we generally can observe four main types of structures:

- **User-item interactions**: usually a matrix, describing implicit (e.g. clicking) or explicit (e.g. rating) interactions between users and items.
- **Consumption sequence**: describing a sequence of consequent consumption events.



(a) User-item bipartite graph.

(b) Sequence graph.

(c) Social relationship between users.

(d) Knowledge graph

Source: "Graph Neural Networks in Recommender Systems: A Survey"

Figure 4.1: Commonly used recommender systems data types can be naturally represented via graph structures.

- **Social relationship graph**: describing an interaction relationship between users.
- **Knowledge graph of an item**: describing properties of items.

From the Figure 4.1 we can indeed see that recommendation data naturally fits graph structures. In fact, user-item interactions can be represented as a user-item bipartite graph, where on one side we have users and on the other we have items, while edges represent consumption/ranking event of an item by an user. Sequences of consumption can be represented as a sequence graph, where nodes represent items and edges represent ordering between the relative consumption events. Social relationships are represented as a social interaction graph, where nodes represent users and edges represent an interaction between them. Finally, the knowledge graph already represents properties of items via a graph.

Graph-based recommendation field provides a vast literature with a rich choice of models for the before-mentioned data types [54][65][68][49]. Authors of [67] survey summarize the main families of models and provide a taxonomy for majority of them. Adopting their taxonomy, subsequent sections of this chapter will be dedicated to describing the applicability of graph neural networks to solve general and sequential recommendation tasks.

4.2 General recommendation

General recommendation makes the assumption that user preferences are invariant over time. This family can be split into subfamilies based on different types of data used. When no side information is used, the models operate on a simple bipartite user-item graph. When social network information is used, the GNN techniques can leverage on social graph's user-to-user data in order to build better user representations. Lastly, item-to-item information of a knowledge graph can be also used by some approaches in order to enhance item representations.

Within this recommendation setting, the rating prediction task could be divided into two subtasks:

- **Transductive Rating Prediction**: users and items appearing in the testing graph are also observed in the training graph. Prior collaborative filtering models would primarily concentrate on this task.
- **Inductive Rating Prediction**: users and items appearing at test time were not necessarily present in the training graph, however, when appearing in test graph, we have access to their ratings, and can use them to make predictions, without retraining the model, as was the case of traditional collaborative filtering models.

GCN based models mainly address the above tasks by learning both transductive and inductive node representations.

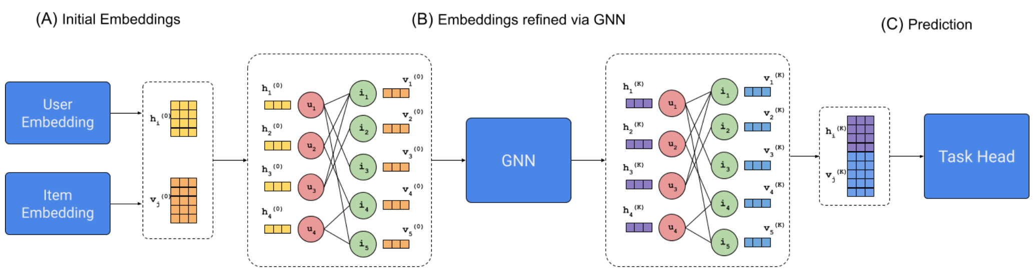


Figure 4.2: General structure of graph-based general recommenders. *(A)* Initial embeddings are produced from user and item ids. *(B)* A Graph Neural Network is used to refine the initial embeddings by aggregating information over node neighbors. *(C)* Embeddings produced via the graph model are passed to the task head which uses them as dense vectors.

4.2.1 Structure

Graph based general recommendation architectures are typically composed of the following parts:

1. **Embeddings**: ids associated to items and users present in the graph are first embedded by an embedding layer to produce initial, dense node representations.

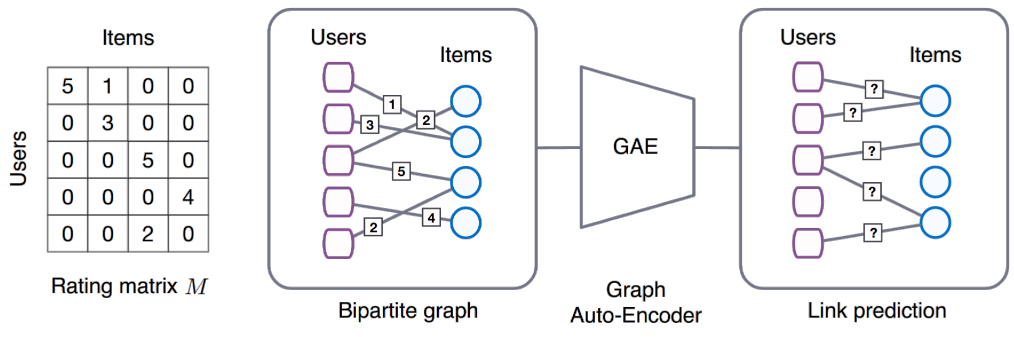
2. **Graph model**: a graph neural network is applied on top of the initial node representations to produce new node representations taking advantage of information being structured as a graph.

3. **Task head**: is applied on top of learned user and item node embeddings together with an appropriate loss (e.g. BPR ranking loss).

General recommendation literature focuses on second part, graph model, of the described components.

4.2.2 Architectures

Further in this section will be a list of selected architectures used for general recommendation. Initially, a pool of 27 models, as found in [67], was considered. To narrow down the amount of architectures, a selection procedure was



Source: "Graph Convolutional Matrix Completion"

Figure 4.3: Rating matrix M of user-items interactions is represented as a bipartite-graph with edges indicating user-preferences. A graph auto-encoder module is used to learn node embeddings from which new, unobserved, edges are reconstructed. This way, the problem is reduced to a link prediction task via an end-to-end trainable graph-autoencoder.

used to consider only the models whose code was publicly available for reproducibility. Additionally, on top of the publicly available code, code that did not run or had no clear usage instructions would lead the model to be ignored in our report.

The reported models will be divided into two categories, based on the type of input graph data: bipartite graph only and social network graph plus bipartite graph, meaning that social network is to be used to augment user representations.

Models for bipartite graph only include: GC-MC, SpectralCF, STAR-GCN, NGCF, LightGCN, LR-GCCF, MCCF, DGCF and DGCF.

While the models that also include social network data are: DiffNet, GraphRec, DANSER and DiffNet++.

GC-MC

Graph Convolutional Matrix Completion (GC-MC) [54] was one of the earliest models for the recommendation task that used graph neural networks. The idea of this work was to solve the matrix completion task from features extracted via a graph auto-encoder module based on differentiable message passing over bipartite user-item interaction graph. As mentioned before, the interaction matrix gets transformed into a bipartite graph, on top of which node embeddings get learned via graph auto-encoder. Those node embeddings, for both items and users, are then used to solve a link prediction task, trainable in an end-to-end fashion.

Practically, initial node embeddings are used by GC-MC via a bilinear decoder, where Q_r is a trainable matrix for each ranking value (i.e. 1 to 5) of shape $E \times E$ and E is dimensionality of user and items node embeddings, to reconstruct the interaction matrix $\hat{M}_{ij}$, in which probabilities of a certain ranking are given by

$$P(\hat{M}_{ij} = r) = \frac{\exp(u_i^\top Q_r v_j)}{\sum_{s\in R} \exp(u_i^\top Q_s v_j)}$$

Negative log likelihood of predicted ratings $\hat{M}_{ij}$ is minimized at the positions of real ratings:

$$\mathcal{L} = - \sum_{i,j;\Omega_{i,j}=1} \sum_{r=1}^{R} I[M_{ij} = r] \log P(\hat{M}_{ij} = r)$$

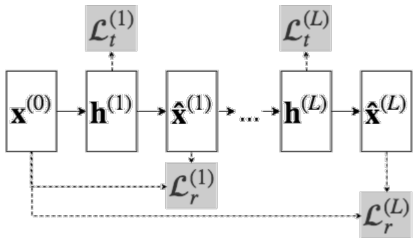
where Ω is a binary matrix that serves as a mask for observed ratings, such that ones occur at indices i, j of observed ratings and zeros of unobserved.

A summary schema for GC-MC architecture is shown in Figure 4.3.

SpectralCF

SpectralCF [72] was another early work that would leverage the spectral convolution operation to directly learn latent users and items factors from spectral domain.

Given X_u and X_i user and item representations respectively, a spectral



Source: “STAR-GCN: Stacked and Reconstructed Graph Convolutional Networks for Recommender Systems”
Figure 4.4: Summary of the STAR-GCN model architecture.

convolution operation to produce new representations could be defined as:

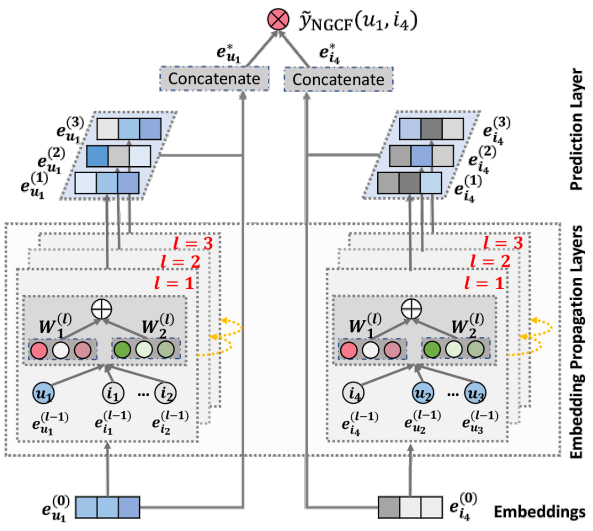
$$\begin{bmatrix} X_u^K \\ X_i^K \end{bmatrix} = \sigma\left((UU^\top + U\Lambda U^\top) \begin{bmatrix} X_u^{K-1} \\ X_i^{K-1} \end{bmatrix} \Theta_{K-1} \right)$$

where Θ is a parameter matrix of the operation, U a matrix of eigenvectors and Λ a vector of eigenvalues of graph laplacian matrix respectively. A block of such spectral convolutions would be stacked together to form a deeper network.

STAR-GCN

STAR-GCN [71] is a multi-block architecture that uses a stack of L GCN encoder-decoder modules, as shown in Figure 4.4. Encoder component is used to generate new latent node representations by encoding neighborhood information together with input node features. Decoder component on the other hand is used to recover input node embeddings given the latent representations produced by encoder. Representations obtained via the encoder are trained on a task-specific loss, while the reconstructions obtained via decoder are trained via a reconstruction loss.

In order to tackle the cold start problem and to learn embeddings that are generalizable to new nodes beyond the train set, the authors propose to use an initial embedding table where a percentage (e.g. 20%) of whole input embeddings is masked at random, by having their values set to zero. Training the network would learn to reconstruct those masked embeddings based on



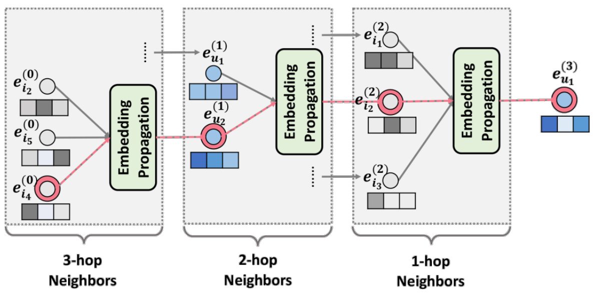
Source: "Neural Graph Collaborative Filtering"

Figure 4.5: Summary of the NGCF model architecture. Initial representations $e^0_{u_1}$ and $e^0_{i_4}$ are refined with multiple embedding propagation layers, this corresponds to a propagation over a 3-hop neighborhood. Produced embeddings are then concatenated together and finally passed to the task head.

neighborhood information. In a testing scenario instead, those embeddings for unknown nodes would be initialized with zeros and iteratively refined by L modules of the network.

In their paper, the authors had also discovered that GCN-based models would leak labels during training. This would occur because the neighborhood aggregation operator is applied on a bipartite graph where ground-truth (known from train data) user-item ratings are used to build the edges, leaking the labels into the graph structure and making the model behave as $r = f_\theta(x, r)$ instead of $r = f_\theta(x)$.

To avoid this leakage issue, authors provide a *sample-and-remove* training strategy, according to which a fixed portion of edges would be removed from training graph at batch sampling time, before using it for training the model.



Source: "Neural Graph Collaborative Filtering"

Figure 4.6: Illustration of a third order embedding propagation for a user u_1.

NGCF

Neural Graph Collaborative Filtering (NGCF) [60] can be seen as a generalization of typical collaborative filtering algorithms that are based on embedding layer plus interaction modeling, for example SVD++. It employs a similar architecture to GC-MC where the graph model however, has a depth of 3, allowing it aggregate information over a 3-hop neighborhood, also called third-order propagation (a better illustration can be seen in Figure 4.6).

User and item embeddings are refined between the hop blocks by an element-wise product between item and user embeddings inside the messages:

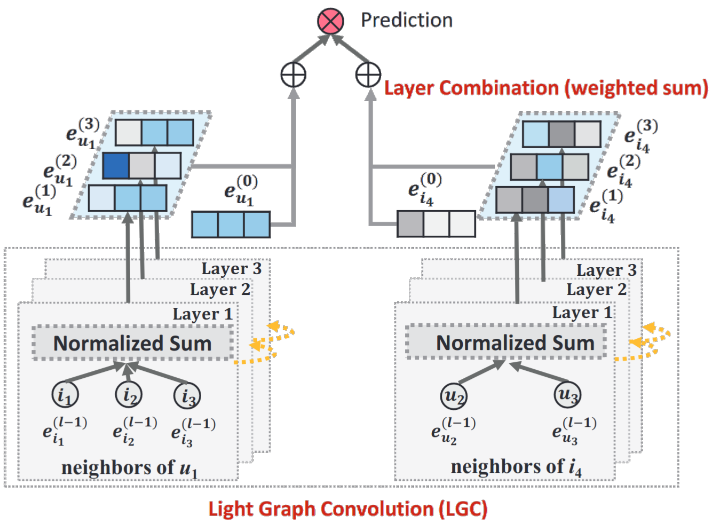
$$m_{u \leftarrow i} = \frac{1}{\sqrt{|\mathcal{N}_u|}\sqrt{|\mathcal{N}_i|}}(W_1 e_i + W_2(e_i \odot e_u))$$

this makes the message dependent on affinity between e_i and e_u, increasing the message passing ability from similar items.

A stack of three such blocks is used and the final embeddings per block are concatenated into a single embedding before being passed to the task head, as seen in summary Figure 4.5.

LightGCN

LightGCN [20] architecture is based on simplifying the design of GCN, which is an essential component for collaborative filtering. Authors simplify the



Source: "LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation"

Figure 4.7: Summary of the LightGCN model architecture.

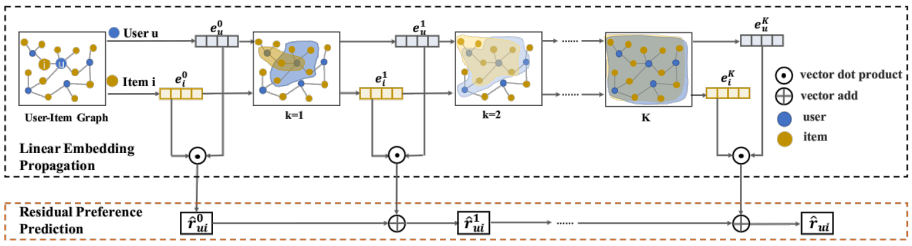
graph convolution operation by removing the non-linearity activation function, reducing the whole model to a single set of parameters to apply, while leaving the neighborhood aggregation component, ending up with the following embedding update formula:

$$e_u^K = \sum_{i\in\mathcal{N}_u} \frac{1}{\sqrt{|\mathcal{N}_u|}\sqrt{|\mathcal{N}_i|}} e_i^{K-1}$$
$$e_i^K = \sum_{u\in\mathcal{N}_i} \frac{1}{\sqrt{|\mathcal{N}_u|}\sqrt{|\mathcal{N}_i|}} e_u^{K-1}$$

where the only trainable parameters of the model are initial embeddings e_u^0 for all users and e_i^0 for all items.

It is worth noting that LightGCN only aggregates neighborhood nodes, without integrating information from the node itself. This choice is made because the self-connections generally need to be treated as a special case of aggregation operation. Additionally, authors show that their proposed feature concatenation strategy used before the task head, already has the same effect as adding self-connections.

Before being passed to task head, features from different layers, functioning as a simple neighborhood aggregator, are combined together as a linear



Source: "Revisiting Graph based Collaborative Filtering: A Linear Residual Graph Convolutional Network Approach"

Figure 4.8: Summary of the LR-GCCF model architecture.

combination:

$$e_u = \sum_{k=0}^{K} \alpha_k e_u^k$$
$$e_i = \sum_{k=0}^{K} \alpha_k e_i^k$$

where α_k is a layer specific parameter that authors set to $1/(K+1)$ for simplicity. A summary of the architecture can be found at Figure 4.7.

LR-GCCF

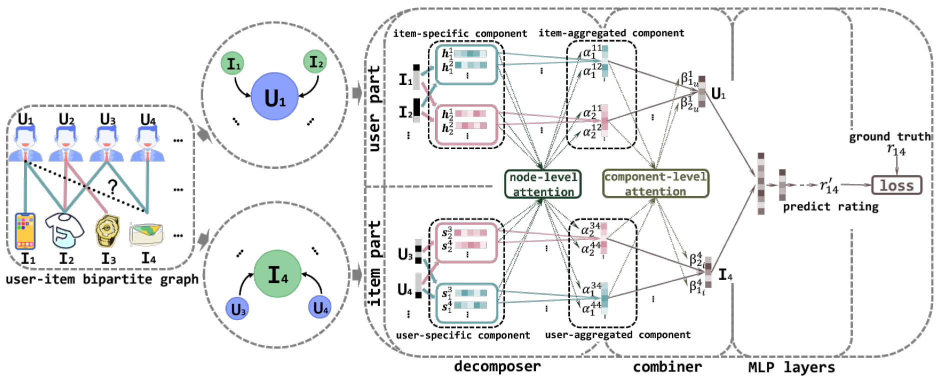
LR-GCCF [8] works similarly to LightGCN, by removing non-linearity, and by using a residual network. The difference with LightGCN however is the aggregation step, which uses self-loops in the embedding update and keeps a weight matrix for each layer:

$$e_u^K = (\frac{1}{|\mathcal{N}_u|} e_u^{K-1} + \sum_{i \in \mathcal{N}_u} \frac{1}{\mathcal{N}_u \mathcal{N}_i} e_i^{K-1}) W^{K-1}$$
$$e_i^K = (\frac{1}{|\mathcal{N}_i|} e_i^{K-1} + \sum_{u \in \mathcal{N}_i} \frac{1}{\mathcal{N}_u \mathcal{N}_i} e_u^{K-1}) W^{K-1}$$

Additionally, residual connections are applied after every layer via:

$$\hat{r}_{ui}^K = \hat{r}_{ui}^{K-1} + < e_u^K, e_i^K >$$

Finally, the representation at last layer, $\hat{r}_{ui}$ gets used as final embedding to be passed to the task head, as seen in summary of the architecture in Figure 4.8.



Source: "Multi-Component Graph Convolutional Collaborative Filtering"

Figure 4.9: Summary of Multi-Component Graph Convolutional Collaborative Filtering (MCCF) architecture. The example shows how final rating prediction for user U_1 consuming item I_4 is calculated via decomposer and combiner modules.

MCCF

Multi-Component Graph Convolutional Collaborative Filtering (MCCF) [62] was a work that aimed at decomposing user intent when consuming a certain item: different users might consume items based on different motivations (e.g. low price items will most likely be purchased by group of people who look for low price rather than by someone who cares more about appearance).

The proposed architecture introduces two blocks to achieve its goals:

1. **Decomposer**: whose role is to decompose graph edges between users and items to identify latent components hidden within the edge/consumption event.

2. **Combiner**: whose role is to recombine latent components to obtain embeddings to be used for predictions.

The main assumption of the architecture is that the user-item interaction graph is driven by M latent components, each responsible for different interaction motivations.

Decomposer block is used to model M different consumption intents of user-item interactions by means of M transformation matrices $\mathbf{W} = \{\mathbf{W}_1, ..., \mathbf{W}_M\}$ and $\mathbf{Q} = \{\mathbf{Q}_1, ..., \mathbf{Q}_M\}$, for users and items respectively. Given an item i, its

m-th latent component can be computed via:

$$\mathbf{h}_m^i = \mathbf{Q}_m \mathbf{e}^i$$

similarly for an user u:

$$\mathbf{s}_m^u = \mathbf{W}_m \mathbf{e}^u$$

Using those two transformations, we can obtain for each item i a set $\{\mathbf{h}_m^i\}_{m=1}^M$ and for each user u a set $\{\mathbf{s}_m^u\}_{m=1}^M$ of embeddings produced by different latent motivations. A key insight here is that a user u does not need to aggregate information about all their consumed items in order to describe the m-th component. This comes from the assumption that each component m is a latent motivation that is responsible only for a subset of total consumptions. The possibility of user u purchasing item i based on the m-th latent component can be formulated via a node-level attention mechanism as:

$$e_m^{ui} = att_{node}(\mathbf{s}_m^u, \mathbf{h}_m^i; m) = \sigma(\mathbf{a}_m^\top \cdot [\mathbf{s}_m^u || \mathbf{h}_m^i])$$

where att_{node} is used to perform node-level attention and can be any deep neural network. This possibility e_m^{ui} is normalized over items via a softmax to produce weights coefficients

$$\alpha_m^{ui} = \text{softmax}(e_m^{ui}) = \frac{\exp{(e_m^{ui})}}{\sum_{j \in \mathcal{N}_u} \exp{(e_m^{uj})}}$$

Finally, m-th item-specific components can be aggregated over all items to learn m-th item-aggregated component z_m^u for user u:

$$\mathbf{z}_m^u = \sigma\left(\sum_{i \in \mathcal{N}_u} \alpha_m^{ui} \cdot \mathbf{h}_m^i\right)$$

So each user u will have M item-aggregated components $\{\mathbf{z}_m^u\}_{m=1}^M$ associated with them, that will be used to learn final user embeddings.

The motivation behind the combiner block is that different latent components, produced by decomposer, can have different contributions towards the learned user embeddings, as the user can have multiple (one or more) different motivations when making a consumption. To tackle the fact that motivation contributions can be different, a component-level attention mechanism is used to learn the importance of different item-aggregated components (component-level attention). Component-level attention takes M item-aggregated user u's components $\{\mathbf{z}_m^u\}_{m=1}^M$ as input and learns to weight each item-aggregated component $\mathbf{z}_m^u$ via:

$$(\beta_1^u, ..., \beta_M^u) = att_{com}(\mathbf{z}_1^u, ..., \mathbf{z}_M^u)$$

where att_{com} is a deep neural network which is used to perform attention at latent component level.

Finally, composer block uses the component-level attention scores to produce final user representation via:

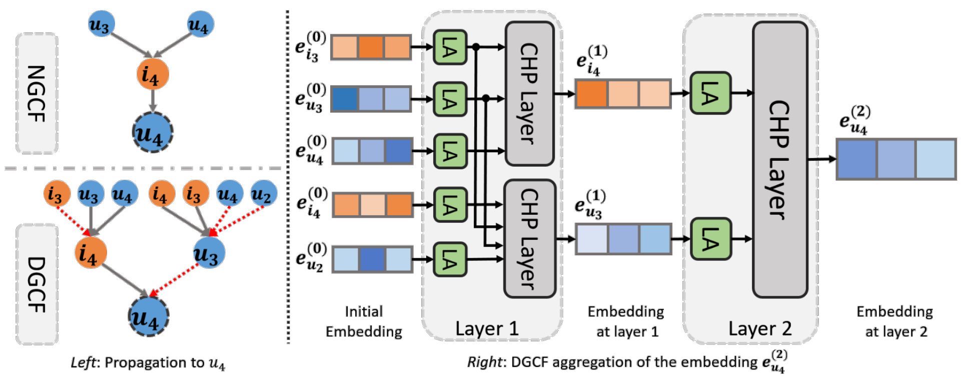
$$\mathbf{z_u} = \sum_{m=1}^{M} \beta_m^u \cdot \mathbf{z_m^u}$$

Item-representation process is performed in a similar way and the summary of the whole architecture can be seen in Figure 4.9.

DGCF

Deoscillated Graph Collaborative Filtering (DGCF) [33] work observes and provides solutions to different problems present in existing graph-based recommenders that use more than one layer:

1. **Oscillation**: problem that occurs when multiple layers are stacked and applied on a bipartite graph. What happens practically is that, for example, 1-hop neighbors of users are all items, while 2-hop neighbors are all users, which implies that aggregating direct neighbors. Consequently,



Source: "Deoscillated Adaptive Graph Collaborative Filtering"

Figure 4.10: *Left:* Propagation of information propagation to node u_4 between NGCF and DGCF architectures. *Right:* DGCF architecture summary.

users only receive information from items and vice-versa.

2. **Varying locality**: is a property of density of local structures in a bipartite graph. It intuitively can be viewed as the total number of nodes that are covered at K-th hop from a given central node, when applying a K-layer network on top of it.

3. **Fixed propagation pattern**: happens when propagation layers induce redundant information to be propagated across multiple layers, multiple times. Due to oscillation, information will eventually get back to the original node and redundantly propagated from it, duplicating the amount of information.

Authors propose to solve the oscillation problem by means of a cross-hop propagation (CHP) layer, that changes the bipartite structure to a regular graph. Node embedding equation becomes:

$$\mathbf{e}_u^{(l)} = \sum_{j \in \tilde{\mathcal{N}}_u} \alpha_j^{(l)} p_j \mathbf{e}_j^{(l-1)}$$

where $\tilde{\mathcal{N}}_u$ is an extended neighbors set that includes a cross-hop neighborhood, p_j is a normalization factor and $\alpha_j^{(l)}$ is an adaptive locality weight coefficient of node j, learned at training time.

Before the CHP layer, DGCF authors place locality-adaptive (LA) layers that adaptively control propagation process for each node via $\alpha_j^{(l)}$ coefficient from the previous equation. Intuitively this layer assigns an influencing factor between 0 and 1 to each node before performing the propagation and which is computed as:

$$\alpha_j^{(l)} = \sigma(\mathbf{w}_{LA}^{(l)})$$

where $\mathbf{w}_{LA}^{(l)} \in \mathbb{R}^{|\mathcal{U}|+|\mathcal{I}|}$ is a trainable parameter vector for l-th LA layer. Another effect of the locality-adaptive layer is the introduction of layer-wise adaptivity, that allows to propagate information in a more efficient, non-redundant way, by acting like a gate.

After the L-th layer propagation, embeddings are averaged together across layers to construct final embeddings used for prediction:

$$\mathbf{E}^* = \frac{1}{L+1}\sum_{l=0}^{L}\mathbf{E}^{(l)}$$

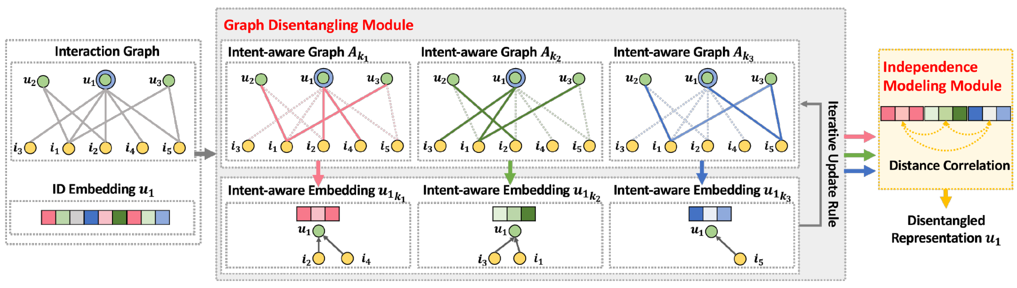
Final summary of the architecture for Deoscillated Graph Collaborative Filtering can be seen in Figure 4.10.

DGCF

Disentangled Graph Collaborative Filtering (DGCF) [61], similarly to MCCF, focuses on disentangling user-item relationships in order to find different factors in user intents. Similarly to MCCF, DGCF uses two key components to achieve such disentanglement:

1. **Graph disentangling module**: whose role is to slice each user and item embedding into chunks, each of which is coupled with an intent, and then provide a routing mechanism into the graph neural network to disentangle the interaction graph.

2. **Independence modeling module**: which acts as a distance correlation regularizer to encourage independence of individual intents.


Graph Disentangling Module
Interaction Graph
ID Embedding u_1
Intent-aware Graph A_{k_1}
Intent-aware Graph A_{k_2}
Intent-aware Graph A_{k_3}
Intent-aware Embedding $u_{1_{k_1}}$
Intent-aware Embedding $u_{1_{k_2}}$
Intent-aware Embedding $u_{1_{k_3}}$
Iterative Update Rule
Independence Modeling Module
Distance Correlation
Disentangled Representation u_1

Source: "Disentangled Graph Collaborative Filtering"

Figure 4.11: Disentangled Graph CF architecture summary. Interaction graph is decomposed into multiple intent-aware graphs, whose adjacency matrices A_{k_i} are learned during training with their relative intent-aware embedding chunks. Final representations for intent-aware embeddings are further decorrelated via a distance correlation loss used in alternation with BPR loss during training.

As a part of graph disentangling module, the embeddings for user/item IDs are separated into M chunks, randomly initialized, each associated with a different intent:

$$\mathbf{u} = (\mathbf{u}_1, ..., \mathbf{u}_M)$$

Authors note that using a single transformation matrix is not sufficient to capture a variety of intents and define $S = \{S_1, ..., S_M\}$, a set of score matrices for M latent intents, where $S_m(u, i)$ denotes interaction between user u and item i due to intent m. Such a set of matrices yields a score vector:

$$S(u, i) = S_1(u, i), ..., S_M(u, i)$$

over M latent intents, initialized as:

$$S(u, i) = (1, ..., 1)$$

assuming an equal intent contribution at the start of training.

As mentioned before, S_m matrices can be seen as adjacency matrices of an intent-aware graph, with $\{\mathbf{u}_k, \mathbf{i}_k | u \in \mathcal{U}, i \in \mathcal{I}\}$ being its initial features.

On such a graph, embeddings can be aggregated by only considering the k-th feature portion related to the appropriate intent, which authors call *intent-aware embeddings*:

$$\mathbf{e}_{ku}^{(l)} = g(\mathbf{e}_{ku}^{(l-1)}, \{\mathbf{e}_{ki}^{(l-1)} | i \in \mathcal{N}_u\})$$

Intent-aware graph's adjacency matrix on the other hand is obtained by a more complicated procedure described in section 3.1.3 of [61] and omitted for simplicity in this work.

Finally, Independence Modeling Module is used to enforce difference between different chunked representations associated with intents, as well as to avoid redundancy between different intents (i.e. to avoid $\mathbf{u}_{k'}$ being reconstructable from other ks different from k'). This can be formulated as a loss term:

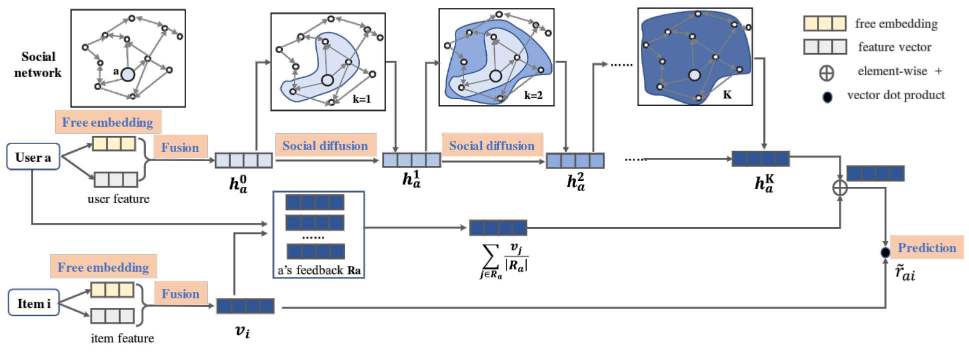
$$loss_{ind} = \sum_{k=1}^{K} \sum_{k'=k+1}^{K} dCor(\mathbf{E}_k, \mathbf{E}_{k'})$$

where $dCor(\mathbf{E}_k, \mathbf{E}_{k'})$ is a distance correlation between chunks k and k' of all embeddings present in the graph. During the model optimization phase, pairwise BPR loss used for the task is alternated with $loss_{ind}$ to provide uncorrelated user and item representations. A more detailed summary on the architecture is available at Figure 4.11.

DiffNet

DiffNet [65] was one of the first approaches that combined traditional user-item bipartite graph with social network data to enhance user representations. Social network data is used to build a model of deep influence propagation, that simulates how users are influenced by recursive social diffusion process (i.e. how users are influenced by other users when consuming items).

The main difference between this and previously explained interaction-graph only models is that here item representations obtained via bipartite graph need to somehow be merged together with user representations present in both



Source: "A Neural Influence Diffusion Model for Social Recommendation"

Figure 4.12: Summary of DiffNet architecture. Side information for both user and item initial representations is produced by a Fusion stage, that takes user/item side features together with their initial embeddings. Afterwards, user's social representation is refined via a Social Diffusion process, while item representations are refined via interaction graph.

bipartite and social graphs.

DiffNet solves the merging problem by means of a Social Diffusion operation, which performs aggregations over social graph to iteratively refine feature representations for user nodes, in a process called social diffusion modeling. Finally, the user embedding to be passed to the task head together with item embeddings, is given by user representations from social diffusion modeling procedure $\mathbf{h}_u^K$ and the sum of normalized item representations consumed by the user:

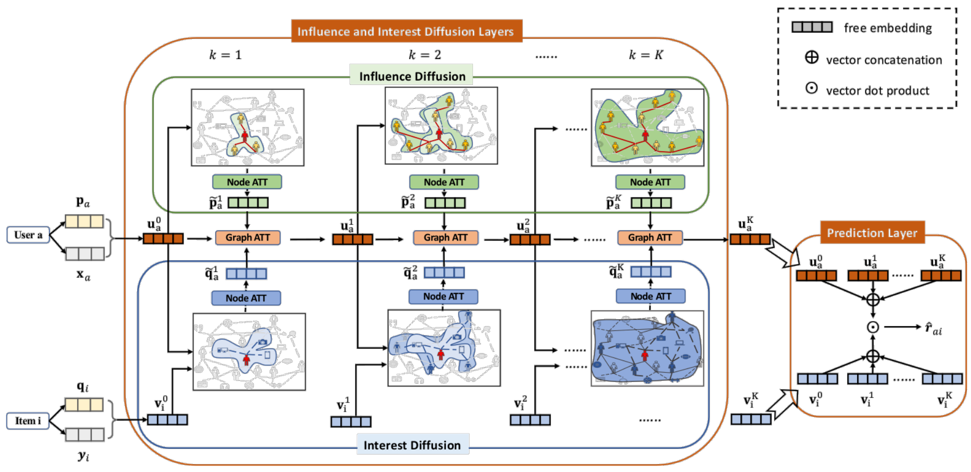
$$\mathbf{e}_u = \mathbf{h}_u^K + \sum_{i \in \mathcal{N}_u} \frac{\mathbf{v}_i}{|\mathcal{N}_u|}$$

where $\mathcal{N}_u$ are the neighbors, representing items consumed by user u in the bipartite graph.

As in former models, final rating, is estimated by taking a product between user and item embeddings:

$$\hat{r}_{ui} = \mathbf{v}_i^\top \mathbf{e}_u$$

Additionally Fusion layers are used to merge side information (e.g. age, gender, income, etc.), together with initial embeddings produced by embedding layers for both users and items. A Fusion layer can be implemented by a simple MLP whose input is a concatenation of side and initial embeddings



Source: "DiffNet++: A Neural Influence and Interest Diffusion Network for Social Recommendation"
Figure 4.13: Summary of DiffNet++ model architecture. Node-level attention is used to assign weights in the aggregation phase in each graph. Graph-level attention is used to fuse interest graph and social graph representations together.

over the feature axis. A visual summary of the model's architecture is provided at Figure 4.12.

DiffNet++

DiffNet++[64] improves the original DiffNet model by introducing attention mechanism and an interest diffusion model to the already existing social diffusion model. Interest diffusion's goal is to exploit propagation on the bipartite interaction graph to learn better item representations, as opposed to only exploiting social graph propagation in the original DiffNet architecture.

As in DiffNet, initial user and item representations, $\mathbf{u}_a^0$ and $\mathbf{v}_i^0$ are produced by combining the relative initial embeddings and their side information via a Fusion procedure.

Interest diffusion procedure is used to learn item representations $\mathbf{v}_i^k$ by performing an aggregation with a depth K on bipartite user-item interaction graph:

$$\tilde{\mathbf{v}}_i^{k+1} = \sum_{a \in \mathcal{N}_i} \eta_{ia}^{k+1} \mathbf{u}_a^k$$

$$\mathbf{v}_i^{k+1} = \tilde{\mathbf{v}}_i^{k+1} + \mathbf{v}_i^k$$

where η_{ia}^{k+1} are attention coefficients between users and items.

User representations on the other hand, are influenced by both social influence diffusion and item interest diffusion procedures. Social diffusion, like in DiffNet, takes initial user representation $\mathbf{u}_a^0$ and performs K aggregation steps over the social graph, computing $\tilde{\mathbf{p}}_a^{k+1}$ with the aid of an user-user attention mechanism:

$$\tilde{\mathbf{p}}_a^{k+1} = \sum_{b \in \mathcal{S}_a} \alpha_{ab}^{k+1} \mathbf{u}_b^k$$

Item contribution to user a's embedding, is computed in a similar fashion by:

$$\tilde{\mathbf{q}}_a^{k+1} = \sum_{i \in \mathcal{N}_u} \beta_{ai}^{k+1} \mathbf{v}_i^k$$

Finally, user's embedding uses both contributions to update current user representation $\tilde{\mathbf{u}}_a^k$ via the following formula:

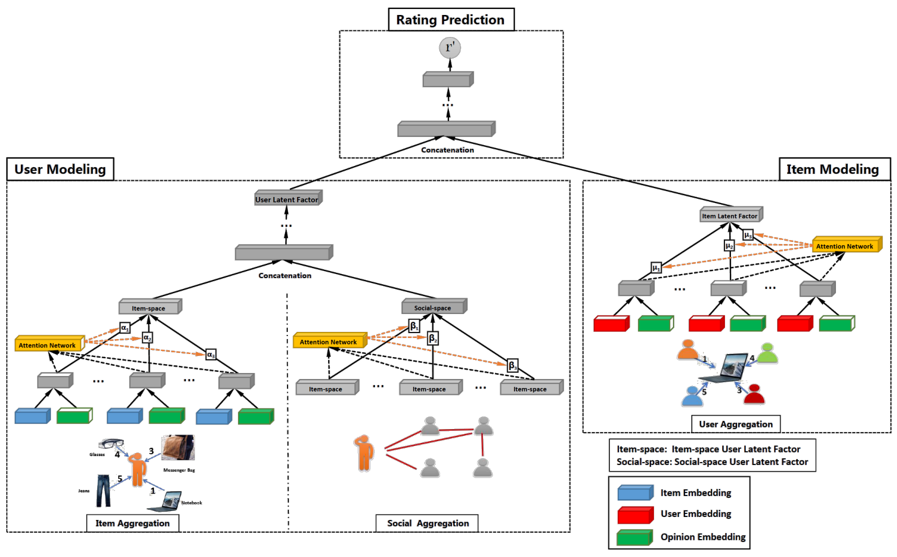
$$\mathbf{u}_a^{k+1} = \mathbf{u}_a^k + (\gamma_{a1}^{k+1} \tilde{\mathbf{p}}_a^{k+1} + \gamma_{a2}^{k+1} \tilde{\mathbf{q}}_a^{k+1})$$

where γ_{a1}^{k+1} and γ_{a2}^{k+1} are graph-level attention coefficients that are learned at training time to weight contributions from influence and interest diffusion procedures for individual users.

Prediction of final rating is ultimately performed by concatenating representations of users and items at each step of diffusion procedures:

$$\begin{aligned}
\mathbf{u}_a^* &= [\mathbf{u}_a^0 || \mathbf{u}_a^1 || ... || \mathbf{u}_a^K] \\
\mathbf{v}_i^* &= [\mathbf{v}_i^0 || \mathbf{v}_i^1 || ... || \mathbf{v}_i^K] \\
\hat{r}_{ai} &= \mathbf{u}_a^{*T} \mathbf{v}_i^*
\end{aligned}$$

Final summary of the architecture can be seen at Figure 4.13.



Source: “Graph Neural Networks for Social Recommendation”

Figure 4.14: Summary of GraphRec architecture, composed by item modeling, user modeling and rating prediction. Both modeling components use attention mechanism to better learn embeddings. User modeling includes a refinement of user embeddings via a social aggregation of features based on social graph.

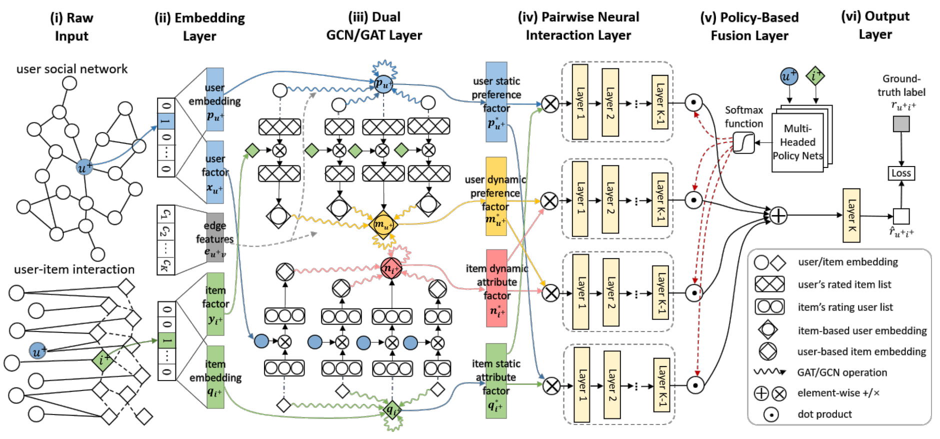
GraphRec

GraphRec [14] was another work that attempted to add a simple social graph extension to the already existing bipartite graph approaches. Compared to DiffNet and DiffNet++ approaches, GraphRec is a more straight-forward extension, as it only adds an additional social aggregation head to the user modeling stage, combining user features generated from both bipartite and social graphs.

Item aggregation is performed as already described in other graph-based recommendation models, with an addition of attention mechanism to better weight item contributions (i.e. the (v_a, u_i) item-user pair):

$$\mathbf{h}_i^I = \sigma\left(\mathbf{W} \cdot \sum_{a \in \mathcal{N}_u} \alpha_{ia}\mathbf{x}_{ia} + \mathbf{b}\right)$$

where α_{ai} is an attention weight computed by a deep neural network and $\mathbf{x}_{ia}$ is an opinion-aware representation for u_i user that takes into consideration v_a



Source: "Dual Graph Attention Networks for Deep Latent Representation of Multifaceted Social Effects in Recommender Systems"

Figure 4.15: Summary of DANSER architecture.

item embeddings and opinion rating dense vector $\mathbf{e}_r$, computed as:

$$\mathbf{x}_{ia} = \mathrm{MLP}(\mathbf{q}_a \oplus \mathbf{e}_r)$$

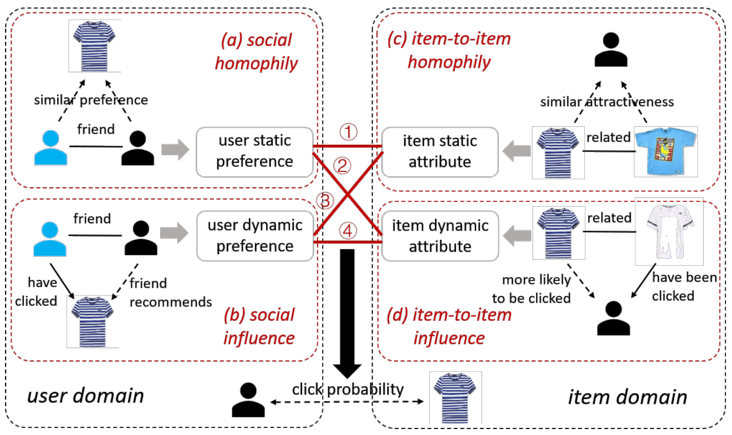
Social aggregation is performed in an analogous way but taking into consideration embeddings of neighbors of users in the social graph, producing a vector for each user u_i denoted by $\mathbf{h}_i^S$.

Lastly, user latent factors (final user embeddings) are computed by passing a concatenation of item aggregation and social aggregation user features to an l-layer MLP:

$$\mathbf{c}_1 = \mathbf{h}_i^S \oplus \mathbf{h}_i^I$$

$$\mathbf{h}_i = \sigma\left(\mathbf{W}_l \cdot \mathbf{c}_{l-1} + \mathbf{b}_l\right)$$

As for the item modeling, item's features are learned via aggregation of user neighbors in an analogous way as item aggregation is done for users, producing final item representations $\mathbf{z}_j$ for item j. A final summary schema of the architecture can be seen in Figure 4.14.



Source: "Dual Graph Attention Networks for Deep Latent Representation of Multifaceted Social Effects in Recommender Systems"

Figure 4.16: Two types of social effects, homophily effect and influence effect that affect user's decision on one item.

DANSER

Dual Graph Attention Networks for Deep Latent Representation of Multifaceted Social Effects in Recommender Systems (DANSER) [66] was yet another early work that attempted to include user social graph information to the already existing bipartite user-item graph schema by including an attention mechanism and a policy-based fusion strategy based on contextual multi-head bandit to weight interactions over social effects.

DANSER works by first introducing an item-item relationship graph, constructed from the bipartite graph by selecting items that are similar in terms of number of users who have clicked both of them, thresholded by a factor. Then it computes user-based item embedding via $\mathbf{x}_u$ user latent factor, item-based user embedding via $\mathbf{y}_i$ item latent factor, user-specific embedding $\mathbf{p}_u$ and item-specific embedding $\mathbf{q}_i$. Next, Dual Graph Attention (GAT) layers are used to capture:

1. **Social homophily**: a static user preference factor $\mathbf{p}_u^*$.
2. **Social influence**: a dynamic, context aware, user preference factor $\mathbf{m}_u^*$.
3. **Item to item homophily**: a static item preference factor $\mathbf{q}_i^*$.

4. **Item to item influence**: a dynamic, context aware, item preference factor $\mathbf{n}_i^*$.

which can be intuitively visualized at Figure 4.16. The four outputs of four GAT layers are then passed to four interaction MLPs, called Pairwise Neural Interaction layers, that compute an interaction score based on different pairwise attributes (denoted as $\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3, \mathrm{s}_4$), as seen in Figure 4.15.

A Policy-based fusion layer then fuses $\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3, \mathrm{s}_4$ into a single representation s, where the sum coefficients are calculated by the means of a stochastic policy gradient algorithm (for further information consult *Policy-Based Fusion Layer* section of [66]).

Finally, the probability estimation of an user u clicking an item i can be estimated as usually by:

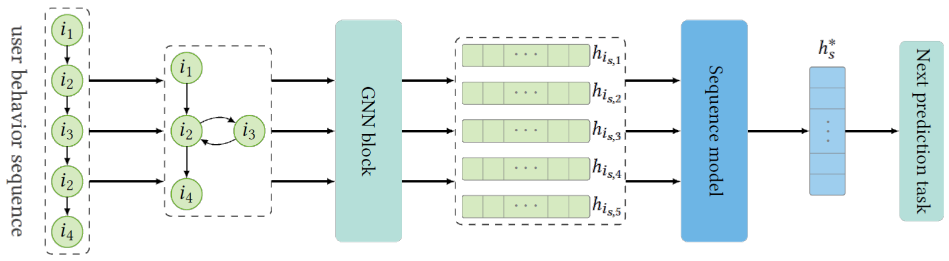
$$\hat{r}_{ui} = \mathrm{MLP}(\mathrm{s})$$

where MLP can be an arbitrarily complex neural network trained using cross-entropy loss for implicit feedback, or mean squared loss for explicit feedback. A complete summary over the model's architecture can be seen in Figure 4.15.

4.3 Sequential recommendation

Sequential recommendation has the core idea to capture transition patterns in items in order to perform the next item recommendation task. Many of the approaches of this family work on behavior sequences produced by users, thus this family of models generally functions by first constructing sequence graphs with the associated item features and then applying a GNN technique on it to produce representations for the prediction task.

Constructing session graphs can be done in different ways, however a common way is to model a session s as a directed graph $\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$ such that each node represents an item $v_{s,i} \in V$, where $V = \{v_1, ..., v_m\}$ is a set of all unique items across all sessions, and $s = [v_{s,1}, ..., v_{s,n}]$ is a list of



Source: "Graph Neural Networks in Recommender Systems: A Survey"
Figure 4.17: General structure of graph-based sequential recommenders.

all the items consumed during the session. Edges are constructed such that $(v_{s,i-1}, v_{s,i}) \in \mathcal{E}_s$ iff user clicks item $v_{s,i}$ after item $v_{s,i-1}$ in the session s.

4.3.1 Structure

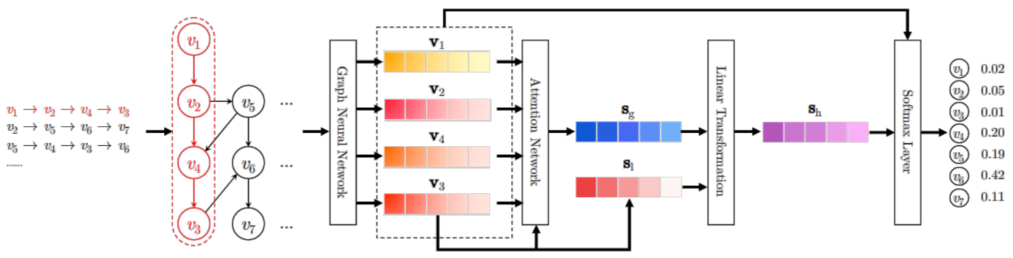
As for the General recommendation case, sequential recommendation approach can be described by a general high-level schema:

1. **Sequence graph generation**: before applying any graph based methods, the sequence data needs to be transformed into an appropriate sequence graph. Different architectures can perform this step differently, for example adding edges between several consecutive items rather than only two consecutive items (i.e. skip connections).

2. **Embeddings**: as sequence data is generally composed of items, their attributes (e.g. id) needs to first be embedded into dense representations to yield better features for the initial layer of the graph model. At this stage, each sequence can be represented by an embedding vector that contains all the embeddings for individual items.

3. **Graph model**: a graph model, generally recurrent, is applied on top of the sequence graph to produce new embeddings for each node in the sequence.

4. **Sequence model**: a model that given all the node embeddings produced



Source: "Session-based Recommendation with Graph Neural Networks"
Figure 4.18: Summary of SR-GNN architecture.

by the graph network, integrates them into a single sequence embedding.

5. **Task head**: a next-item classification head that outputs next item consumption probability, given the whole sequence embedding.

In Figure 4.17 we can see a visualization of the general high-level schema.

4.3.2 Architectures

Similarly to General Recommendation section, an initial pool of 16 models was analyzed, from which only 4 models which had a runnable code were selected: SR-GNN, DGRec, MGNN-Spred and TAGNN.

SR-GNN

Session-based Recommendation with Graph Neural Networks (SR-GNN) [68] was one of the earlier works that attempted to apply graph neural networks to the problem of sequential recommendation.

To produce embeddings $\mathbf{v}_i^t$ for node $v_{s,i}$ of a given graph $\mathcal{G}_s$, SR-GNN performs the following gated GNN [29] update computation:

$$\begin{aligned}
\mathbf{a}_{s,i}^t &= \mathbf{A}_{s,i:}[\mathbf{v}_1^{t-1}, ..., \mathbf{v}_n^{t-1}]^\top \mathbf{H} + \mathbf{b}, \\
\mathbf{z}_{s,i}^t &= \sigma(\mathbf{W}_z \mathbf{a}_{s,i}^t + \mathbf{U}_z \mathbf{v}_i^{t-1}), \\
\mathbf{r}_{s,i}^t &= \sigma(\mathbf{W}_r \mathbf{a}_{s,i}^t + \mathbf{U}_r \mathbf{v}_i^{t-1}), \\
\tilde{\mathbf{v}_i^t} &= tanh(\mathbf{W}_o \mathbf{a}_{s,i}^t + \mathbf{U}_o(\mathbf{r}_{s,i}^t \odot \mathbf{v}_i^{t-1})), \\
\mathbf{v}_i^t &= (1 - \mathbf{z}_{s,i}^t) \odot \mathbf{v}_i^{t-1} + \mathbf{z}_{s,i}^t \odot \tilde{\mathbf{v}_i^t}
\end{aligned}$$

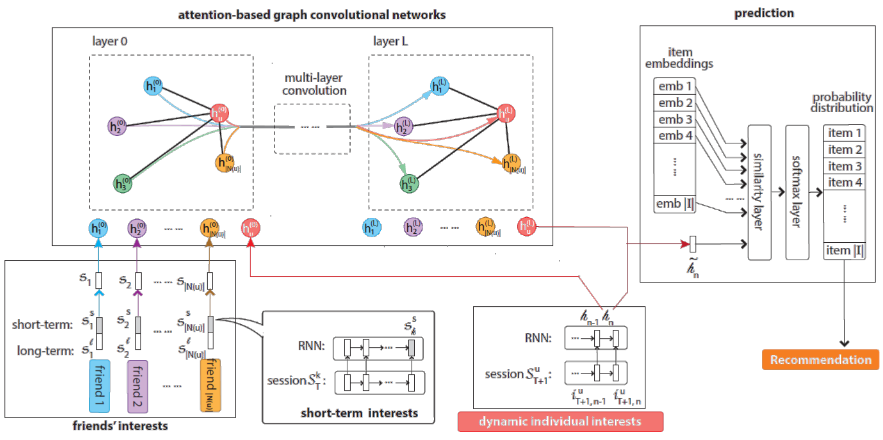
where $\mathbf{A}$ is a connection (a variant of adjacency) matrix, $\mathbf{H} \in \mathbb{R}^{d \times 2d}$ a weight matrix, $\mathbf{z}_{s,i}^t$ and $\mathbf{r}_{s,i}^t$ are reset and update gates respectively. Intuitively, this update procedure first extracts latent vectors for item neighborhoods and feeds them to update and reset gates, which decide which information should be preserved and which should be discarded. Using previous state, current state and reset state, a candidate state is constructed. Finally, the final state is a combination of previous hidden state and candidate state, controlled by the update gate.

After feeding all the session graphs to the graph model, we obtain feature vectors of all the nodes. To represent each session as a single embedding vector $\mathrm{s}_h \in \mathbb{R}^d$ we first compute local and global session embeddings and combine them together via a simple transformation $\mathrm{s}_h = \mathrm{MLP}([\mathrm{s}_l; \mathrm{s}_g])$. Local embedding s_l can be simply defined as the embedding vector $\mathbf{v}_n$ of the last-clicked item $v_{s,n}$. Global embedding on the other hand aggregates all the node vectors of the session graph with the aid of an attention mechanism:

$$\mathbf{s}_g = \sum_{i=1}^{n} \alpha_i \mathbf{v}_i$$

where α_i are attention coefficients between i-th and n-th node embeddings in the sequence.

Finally, the probability distribution over the next item can be computed



Source: "Session-based Social Recommendation via Dynamic Graph Attention Networks"

Figure 4.19: Summary of DGRec architecture. A shared recurrent neural network is used to initially compute dynamic user's interests (i.e. what are the user's interests based on their sessions), short-term and long-term friend's interests. Computed features are then used as node features for a social user graph and using a graph attention neural network, final representations are produced for the current user. This final representation, together with learned item embeddings can be used to perform the next item prediction task.

via:

$$\hat{\mathbf{z}}_i = \mathbf{s}_h^\top \mathbf{v}_i$$

$$\hat{\mathbf{y}} = \text{softmax}(\hat{\mathbf{z}})$$

where $\hat{\mathbf{z}} \in \mathbb{R}^m$ denotes the recommendation logits over all the candidate items. A summary of SR-GNN approach can be seen in Figure 4.18.

DGRec

Session-based Social Recommendation via Dynamic Graph Attention Networks (DGRec) [49] was one of the earlier works that had introduced the usage of social graphs inside session based graph recommendation. The architecture itself has different components it needs to model: friends' long and short term interests, user's dynamic interests.

Dynamic user behaviour is modeled using an RNN that captures the rapidly changing interests by inferring the representation of user's session, $S^u_{T+1} =$

$\{i^u_{T+1,1}, ..., i^u_{T+1,n}\}$ token by token, in a recursive manner via:

$$h_n = f(i^u_{T+1,n}, h_{n-1})$$

where T is the index of current user's session, h_n a representation of user's interests and f a nonlinear function, such as LSTM, that combines item and interest information.

Friends' short-term interests are modeled by taking a subsequence of their recently consumed items (e.g. friend's last session), $S^k_T = \{i^k_{T,1}, i^k_{T,2}, ..., i^k_{T,k}\}$ and applying an RNN on top of it with weights shared from dynamic user's behavior recurrent model:

$$s^s_k = r_{N_{k,T}} = f(i^k_{T,N_{k,T}}, r_{N_{k,T-1}})$$

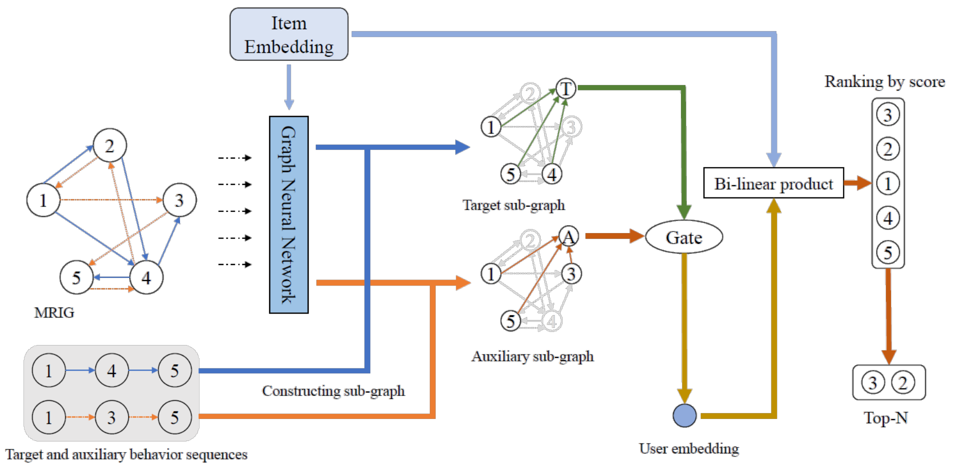
Friends' long-term preferences reflect their average interests and are not time-sensitive, hence why they can be represented by a single vector taken from k-th row of user embedding matrix $\mathbf{W}_u$:

$$s^l_k = \mathbf{W}_u[k, :]$$

Friend's k short and long-term preferences are finally combined by applying a non-linear transformation:

$$s_k = \text{ReLU}(\mathbf{W}[s^s_k; s^l_k])$$

Previous modeling part included only recurrent neural networks, social connectivity data on the other hand, is processed by using a graph-attention neural network to dynamically infer the influencers based on users' current interests. Initial features assigned to graph nodes are given from interests modeling phase, formally $h^{(0)}_u = h_n$ and $\{h^{(0)}_k = s_k, k \in \mathcal{N}_u\}$. After applying a stack of L graph-attention layers, the social-influenced user's representation



Source: "Beyond Clicks: Modeling Multi-Relational Item Graph for Session-Based Target Behavior Prediction"

Figure 4.20: Summary of MGNN-SPred architecture. First, a Multi-Relational Item Graph (MRIG) gets constructed from all user's target and auxiliary behavior sequences. Propagation procedure over MRIG by a GNN yields final item representations to be used when building sequence representations. Sequence representations are computed by mean-pooling over item representations, given from last GNN's propagation step, of items appearing in input target and auxiliary behavior sequences.

is denoted by $h_u^{(L)}$.

Finally, user's final representation is computed by taking into account both recent behaviors h_n and social influences $h_u^{(L)}$, processed by a fully-connected layer:

$$\hat{h}_n = \mathbf{W}[h_n; h_u^{(L)}]$$

The probability over next item is given by taking the softmax function of a dot product between $\hat{h}_n$ and the items' embeddings. A summary of the whole architecture can be seen in Figure 4.19.

MGNN-Spred

Multirelational Graph Neural Network model for Session-based target behavior Prediction (MGNN-Spred) [59] paper has introduced a way to include auxiliary behavior information on top of commonly used target behavior. To accomplish their goal, authors build a Multi-Relational Item Graph (MRIG), based on all behavior sequences from all sessions, including target and auxiliary behaviors. On top of MRIG, a graph neural network is used to learn global item-to-item relations and obtain user preferences w.r.t. current target and auxiliary sequences.

Multi-Relational Item Graph is constructed from a given set of sessions S, which contains both target and auxiliary behavior sequences for the current user. The construction algorithm proceeds to browse all behavior sequences, collect all their items as nodes of the graph and add directional edges between two consequent items if their are present in target or auxiliary behaviors, distinguishing between the edge type (for more details, consult Algorithm 1 of [59]).

The model starts by computing initial item embeddings via learnable embedding tables, and using those embeddings as initial node features for the MRIG. For each node v inside MRIG, there are four sets of neighbors based on type and direction: “target-forward”, “target-backward”, “auxiliary-forward”, “auxiliary-backward”. The previously mentioned sets are defined as:

$$\mathcal{N}_{t+}(v) = \{v' | (v', v, \text{target}) \in \mathcal{E}\}, \mathcal{N}_{t-}(v) = \{v' | (v, v', \text{target}) \in \mathcal{E}\}$$

$$\mathcal{N}_{a+}(v) = \{v' | (v', v, \text{auxiliary}) \in \mathcal{E}\}, \mathcal{N}_{a-}(v) = \{v' | (v, v', \text{auxiliary}) \in \mathcal{E}\}$$

At each step of representation propagation, GNN aggregates each group of neighbors by performing mean-pooling, obtaining item representations from the relative neighbor groups: $\mathbf{h}^k_{t+,v}$, $\mathbf{h}^k_{t-,v}$, $\mathbf{h}^k_{a+,v}$, $\mathbf{h}^k_{a-,v}$. The four representations then get combined together by sum-pooling:

$$\tilde{\mathbf{h}}^k_v = \mathbf{h}^k_{t+,v} + \mathbf{h}^k_{t-,v} + \mathbf{h}^k_{a+,v} + \mathbf{h}^k_{a-,v}$$

and used to update the representation of center node v:

$$\mathbf{h}^k_v = \mathbf{h}^{k-1}_v + \tilde{\mathbf{h}}^k_v$$

After performing K iterations of such propagation procedure, final item representations are given by the last step's representations: $\mathbf{g}_v = \mathbf{h}_v^K$.

Having computed item representations from the MRIG, the model takes the behavior sequence representation as a mean-pooling of representations of items from both target behavior sequence P and auxiliary behavior sequence Q:

$$\mathbf{p} = \frac{\sum_{i=1}^{|P|} \mathbf{g}_{p_i}}{|P|}, \mathbf{q} = \frac{\sum_{i=1}^{|Q|} \mathbf{g}_{q_i}}{|Q|}$$

Authors notice that the contributions of p and q towards next item prediction might be different (e.g. some users might browse item pages frequently and click all the items, while others might only click on items they want to buy), thus introduce a gating mechanism to compute a relative importance weight α:

$$\alpha = \sigma(\mathbf{W}_g[\mathbf{p}; \mathbf{q}])$$

Finally, user preference representation $\mathbf{o}$ of the current session is computed as a weighted summation of p and q:

$$\mathbf{o} = \alpha \cdot \mathbf{p} + (1 - \alpha) \cdot \mathbf{q}$$

Recommendation scores for each item $v \in \mathcal{V}$ are calculated by a bi-linear matching schema between item embeddings $\mathbf{e}_v$ and user representation $\mathbf{o}$:

$$s_v = \mathbf{o}^\top \mathbf{W} \mathbf{e}_v$$

and passed to a softmax, to compute the final probability distribution over the next item, as already mentioned for previous sequential models. A summary of the whole architecture can be seen in Figure 4.20.

TAGNN

Target Attentive Graph Neural Networks (TAGNN) [70] can be seen as an extension of SR-GNN model, where an additional representation, called "target embedding" is included into the final session embedding.

Final session embedding is computed as:

$$\mathbf{s}_t = \text{MLP}([\mathbf{s}^t_{target}; \mathbf{s}_l; \mathbf{s}_g])$$

where $\mathbf{s}_l$ and $\mathbf{s}_g$ are computed exactly as in SR-GNN. Target embedding $\mathbf{s}^t_{target}$ on the other hand uses an attention mechanism to calculate soft attention scores over all items in the session with respect to the target item:

$$\beta_{i,t} = \text{softmax}(e_{i,t}) = \frac{\exp(\mathbf{v}_t^\top \mathbf{W} \mathbf{v}_i)}{\sum_{j=1}^{m} \exp(\mathbf{v}_t^\top \mathbf{W} \mathbf{v}_j)}$$

and the embedding itself represents user's interests towards a target item:

$$\mathbf{s}^t_{target} = \sum_{i=1}^{s_n} = \beta_{i,t} \mathbf{v}_i$$

which varies with different target items.

4.4 Models comparison

In the previous section we've listed various approaches used for general and sequential graph recommendation tasks. In this section we will summarize their results in terms of accuracy metrics across different academic datasets in their domain, as reported in the relative model papers. For the general recommendation task, the reported academic datasets are MovieLens, Amazon-review, Gowalla and Yelp. For the sequential recommendation task, the reported academic datasets are Yoochoose and Diginetica.

MovieLens dataset is composed of users, rated items and the relative rating from 1 to 5. It is distributed in different versions based on number of ratings:

- **MovieLens 100K**: 943 users, 1682 items and 100000 ratings.
- **MovieLens 1M**: 6040 users, 3706 items and 1000209 ratings.
- **MovieLens 10M**: 69878 users, 10677 items and 10000054 ratings.

Table 4.1 illustrates the results of different models on MovieLens datasets. As we can see, Deoscillated Graph Collaborative Filtering approach is the best model for this dataset in terms of recall@20 and ndcg@20 metrics, among the ones described in previous sections.

Amazon-review dataset [35] includes different categories of goods that users have rated throughout 11 years of interactions. In the subsequent results reporting we will consider the following categories:

- **Amazon Book**: 52643 users, 91599 items and 2984108 interactions.
- **Amazon Movies and TV**: 2114748 users, 150334 items and 6174098 interactions.

Table 4.2 illustrates the results of different models on Amazon datasets. As we can see, Deoscillated Graph Collaborative Filtering is still the best model for Amazon Movies and TV dataset in terms of recall@20 and ndcg@20 metrics, while LightGCN is the best model for Amazon Book dataset in terms of same metrics.

Gowalla [30] and *Yelp2018* are two datasets composed of user check-ins in different locations (e.g. user has visited a restaurant). Those datasets are composed of:

- **Gowalla**: 29858 users, 40981 items and 1027370 interactions.
- **Yelp2018**: 31668 users, 38048 items and 1561406 interactions.

Table 4.3 illustrates the results of different models on Gowalla and Yelp2018 datasets. LightGCN results to be the best model across the two datasets in terms of recall@20 and ndcg@20 metrics.

Yoochoose and *Diginetica* are the two datasets on which sequential graph recommendation results are illustrated. The original Yoochoose dataset contains click stream data from an e-commenrce site is composed of 7981580 sessions and 43097 unique items. Yoochoose 1/64 dataset is a version of Yoochoose that uses 1/64 fraction of most recent training sessions for training. Diginetica is another dataset that contains 204771 sessions and 43097 items. The results of sequential graph recommendation models is reported on Table 4.4, from which we can see that TAGNN performs the best on Yoochoose 1/64 and Diginetica datasets, while MGNN-Spred performs the best on the whole Yoochoose dataset.

	MovieLens 100K			MovieLens 1M			MovieLens 10M		
Model	RMSE	recall@20	ndcg@20	RMSE	recall@20	ndcg@20	RMSE	recall@20	ndcg@20
GC-MC	0.910	0.2966	0.1883	0.832	0.2611	0.2069	0.777	-	-
STAR-GCN	**0.895**	-	-	0.832	-	-	**0.770**	-	-
NGCF	-	0.3146	0.1978	-	0.2693	0.2164	-	-	-
LightGCN	-	0.3399	0.2137	-	0.2888	0.2334	-	-	-
MCCF	0.907	-	-	-	-	-	-	-	-
DGCF (Deosc.)	-	**0.3536**	**0.229**	-	**0.3075**	**0.2501**	-	-	-

Table 4.1: Accuracy metrics RMSE (↓), recall@20 (↑), ndcg@20 (↑) for different general recommendation models on different versions of MovieLens datasets.

	Amazon Book		Amazon TV	
Model	recall@20	ndcg@20	recall@20	ndcg@20
GC-MC	0.0288	0.0224	0.0578	0.0475
NGCF	0.0337	0.0261	0.1117	0.0886
LR-GCCF	0.0341	0.0258	-	-
LightGCN	**0.0411**	**0.0315**	0.113	0.0893
DGCF (Disen.)	0.0399	0.0308	-	-
DGCF (Deosc.)	**-**	**-**	**01351**	**0.1083**

Table 4.2: Accuracy metrics recall@20 (↑), ndcg@20 (↑) for different general recommendation models on different versions of Amazon datasets.

4.5 Model categorization

Models described in previous sections were divided in general and sequential families, and characterized by the type of data they used: bipartite graph,

	Gowalla		Yelp 2018	
Model	recall@20	ndcg@20	recall@20	ndcg@20
GC-MC	0.1395	0.1204	0.0464	0.0379
NGCF	0.1569	0.1327	0.0579	0.0477
LR-GCCF	0.1518	0.1259	-	-
LightGCN	**0.183**	**0.1554**	**0.0649**	**0.053**
DGCF (Disen.)	0.1794	0.1521	0.064	0.0522
DGCF (Deosc.)	0.1707	0.1384	-	-

Table 4.3: Accuracy metrics recall@20 (↑), ndcg@20 (↑) for different general recommendation models on Gowalla and Yelp2018 datasets.

	Yoochoose		Yoochoose 1/64		Diginetica	
Model	recall@100	mrr@100	precision@20	mrr@20	precision@20	mrr@20
SR-GNN	21.262	2.6892	70.57	30.94	50.73	17.59
TAGNN	-	-	**71.02**	**31.12**	**51.31**	**18.03**
MGNN-Spred	**28.632**	**3.6564**	-	-	-	-

Table 4.4: Accuracy metrics recall@K (↑), mrr@K (↑), precision@K ↑ for sequential recommendation datasets.

bipartite graph + social network, or sequence graph. Now instead, we are interested in categorizing those models, based on criteria that might possibly have an influence on their performance: initial features, neighbor sampling type and model size.

Initial features are the type of initial features the model works with. We have seen in Chapter 2 that Collaborative Filtering recommenders are models that make use of *user and item ids* only, without dealing with features related to their content, which in practice yields to huge embedding tables that might be related to majority of model's parameters count; sequential recommenders, in their simplest form, also make use of item id only. Additional types of features can be *side and multi-modal*, which despite both being item or user content based features, are data of a different nature. Side information is just information that can be included from already existing data when possible, without requiring complex processing. Multi-modal features on the other hand, are audio, textual or image data, that requires specialized processing by using another type of neural network, e.g. BERT for text data, ResNet for image data and so on. The latter suggests that we can either generate embeddings for

Model	Initial Features	Neighbor Sampling
DGCF (deosc)	User id + Item id	All neighbors
GC-MC	User id + Item id + Side	Drop non-batch nodes
NGCF	User id + Item id	All neighbors
GraphRec	User id + Item id	All neighbors
SR-GNN	Item id	All neighbors
LightGCN	User id + Item id	All neighbors
DGRec	User id + Item id	K neighbors
DiffNet	User id + Item id	All neighbors
SpectralCF	User id + Item id	All neighbors
DANSER	User id + Item id	K neighbors
LR-GCCF	User id + Item id	All neighbors
TAGNN	Item id	All neighbors
DGCF (disen)	User id + Item id	All neighbors
LESSR	User id + Item id + Side	All neighbors
DiffNet++	User id + Item id + Multimodal	All neighbors
MGNN-Spred	Item id	All neighbors
MCCF	User id + Item id	K neighbors

Table 4.5: Categorization of models based on the type of initial node features and type of neighbor sampling they perform.

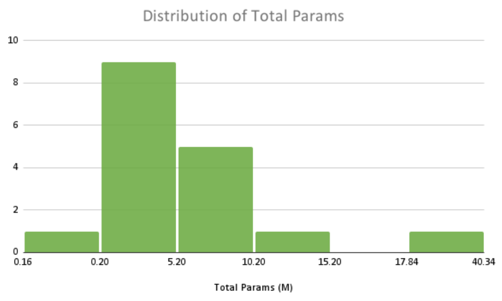
Model	Total Params (M)	Embedding Params (M)	% Embedding
DGCF (deosc)	4.8	4.5	94.1
GC-MC	40.3	40.3	99.9
NGCF	9.3	9.2	99.6
GraphRec	12.2	12.1	99.1
SR-GNN	3.9	3.7	95.9
LightGCN	4.5	4.5	100.0
DGRec	4.0	3.9	96.8
DiffNet	1.8	1.8	99.7
SpectralCF	0.2	0.2	100.0
DANSER	2.0	1.9	93.0
LR-GCCF	9.2	9.2	100.0
TAGNN	3.9	3.7	95.6
DGCF (disen)	9.2	9.2	100.0
LESSR	0.2	0.1	52.7
DiffNet++	5.5	3.6	65.2
MGNN-Spred	3.4	3.4	99.9
MCCF	8.4	6.7	80.1

Table 4.6: Parameter counts per model, considered by taking the biggest value available when running authors' code, number of parameters related to embedding layers, and their percentage w.r.t. total parameters.

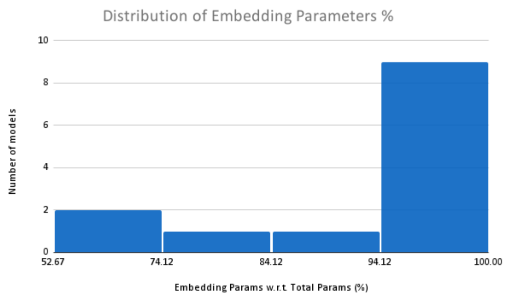
multi-modal features offline, before training, and use some adaptation layer, or to include a whole new finetuneable model to our existing graph model; both of the previous options affect the final model size. We can see in Table 4.5 that majority of the analyzed models uses only user and item ids, despite being trained on datasets that offer additional features.

Neighbor Sampling is the type of sampling that is performed to generate the set of neighbors that a node will aggregate information from, during the aggregation stage of graph layers, which has both influence over memory and time of the layer. In the analyzed models, three types of strategies were found: all neighbors, thresholding and removing nodes. The all neighbor strategy consists in taking all neighbors of a node when performing the aggregation. In order to do so, all the nodes need to be first extracted from the graph, to build a message passing tree (Chapter 3, Figure 3.4), before even starting the calculation of graph layer. Thresholding on the other hand consists in sampling only a subset of K neighbors that will be used in the aggregation phase. There are various strategies of sampling this subset of K neighbors that can affect both accuracy and performance of the model. Omitting the accuracy aspect, the performance gain is motivated by the fact that the model will need to aggregate less information, thus reducing both spatial and temporal cost of the operation. However, the latter statement fails to omit the practical consideration that sampling itself has a runtime cost that could affect the overall performance of the training (the complexity is moved to data-loading part). Finally, one of the analyzed models, GC-MC, in its vectorized implementation, proposed to drop all the nodes that weren't connected to the extremities of sampled edges. Table 4.5 shows the usage of neighbor sampling in various models, from which we can see that majority of models consider all neighbors when performing the aggregation, which could possibly have an impact on their scalability.

Model sizes is the count of total parameters that the model uses. For the analyzed models, we have considered the biggest possible parameters count



(a)



(b)

Figure 4.21: (a) Distribution of total parameters count, in millions, by the number of models. (b) Distribution of embedding parameters count as a percentage of total parameters count

the model would yield, when running with a default configuration provided by the authors. This is due to the fact that collaborative filtering recommendation models have their parameter count highly dependent on the used dataset. In fact, we can see in Figure 4.21b that majority of analyzed models' parameters are composed of mostly embedding parameters, which means that graph based models are not very different in terms of parameters behavior from their deep learning counter-parts. For comparison, NCF, a deep learning based recommender as taken from NVIDIA's examples repository has 20 million parameters, of which 19.8 million are due to embeddings. It is also worth noting from Figure 4.21a and more specifically from Table 4.6 that majority of analyzed models have their parameters count between 1 and 10 millions, with only GC-MC being an exception.

Chapter 5

Experiments

In the previous chapter we have seen that most of the studied graph based recommendation models are of collaborative filtering nature (e.g. they use only user and item ids), have a parameters count in orders from 1 to 10 millions, majority of which is given by embedding layers' parameters. In this chapter, we will discuss the experiments we conducted to study a set of important performance properties of a graph based recommender, implemented in DGL [58], an open source library[1] for graph neural networks. For our experiments we considered a simple graph model, not based on those of Chapter 4. The choice of a simple model can be motivated by:

1. The similarity in the analyzed model's performance-affecting parameters as seen in Chapter 4.

2. DGL library still being in development and not supporting, or inefficiently supporting features that might be used in more complicated models. Implementing those features might not always be an efficient solution in both engineering effort and final runtime.

3. When discovering possible performance inefficiencies, a complex model can be always ablated down to a simpler one (e.g. if majority of the

[1] `https://www.dgl.ai/`

model's time is spent on sampling of graph batches, it does not matter how efficient the model is).

4. It is easier to implement desired features such as multi-GPU data parallelism or support of different datasets when working with a simpler codebase.

which could serve as a proxy to understand the performance of more complicated models. When studying performance we considered the following aspects:

1. Different embedding dimensions.
2. Different batch sizes.
3. Full neighbor sampling vs K neighbor sampling.
4. Single-GPU vs Multi-GPU setup (i.e. 1 vs 8).
5. Different type of precision (i.e. AMP vs FP32/TF32).

from which we have selected a subset of parameters to reduce the overall dimensionality of the studied problem.

5.1 Model definition

For our experiments we chose the MovieLens latest-full dataset, which at the time of writing this work is composed of 280000 users, 58000 movies and 27 million ratings. The choice of this dataset allowed us to experiment with relatively big batch sizes, without risking to run out of data.

The model's architecture is shown at Figure 5.1. It mimics the already seen graph based architectures for collaborative filtering recommendation, with the simplification in graph processing module. In our case, in fact, the graph processing module is composed of three standard graph attention layers, included

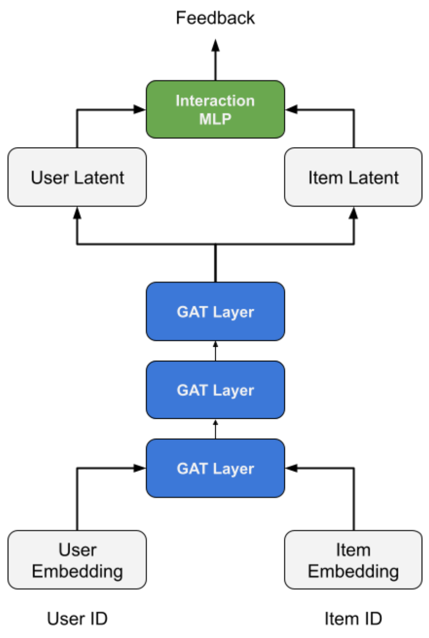


Figure 5.1: Simplified GNN architecture that we will be using. The architecture consists of three stacked graph attention layers used to produce user and item latent features from user and item embeddings. An MLP is then placed on top of user and item latent features to predict feedback from them.

Embedding Dim.	#Parameters (M)
16	5.4
32	**10.8**
64	21.6
128	43.1

Table 5.1: Number of parameters as a function of embedding dimension.

by default in the DGL library. Besides this change, the overall model schema remains the same.

In Table 5.1 we can see how the parameters count varies as a function of embedding dimension used by the model's embedding layers and first GAT layer. To make our study coherent with architectures described in Chapter 4, we have used the embedding dimension of 32, which has yielded us a model with around 10.8 million parameters.

Throughput measure

Throughput can be defined as the amount of data a model can process over a given unit of time (e.g. number of samples per second):

$$\text{Throughput} = \frac{\#\text{Samples}}{Time(\text{Samples})}$$

Generally, when taking the measure, we should let the model "warmup" first, as initial steps of a model can require the framework to initialize its internal structures, significantly alternating the total time taken to process the initial samples. In our study, we considered the throughput for the training phase, which includes batch loading, forward, backward and optimization phases.

5.2 Effects of batch size and sampling

We started our study by finding the appropriate parameters for batch size and neighborhood sampling limit factor. Doing so helped us to greatly limit the dimensionality of the parameter space in the subsequent experiments, letting us skip unwanted parameter combinations.

We had considered the following parameter space:

- **Batch size**: 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304.
- **Neighbor sampling limit**: all neighbors, 50 neighbors, 25 neighbors.

and performed a training of the model for each of the parameter combinations, measuring the throughput at the end of first epoch in a single V100 32GB GPU and FP32 precision setup.

In Figure 5.2 we can see the GPU memory occupation in GiB when performing the training with various batch sizes. It is measured considering full neighborhood sampling and thus can give us an estimation of the upper-bound of memory usage for a particular batch size (i.e. when using limited neighbors

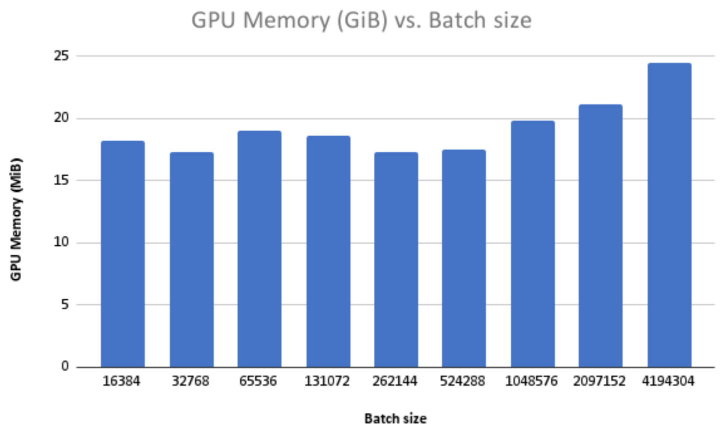


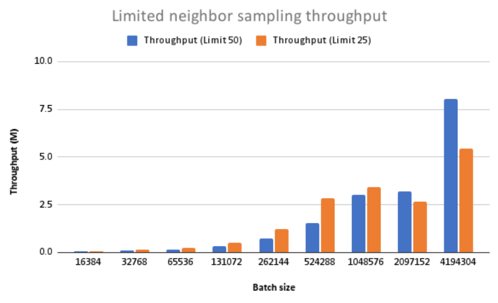
Figure 5.2: GPU memory usage at training time for different batch sizes.

Batch size	Throughput (full)	Throughput (Limit 50)	Throughput (Limit 25)
16384	1427	44867	67482
32768	4107	87384	133141
65536	7105	162641	252807
131072	13493	318144	513199
262144	19146	724127	1250266
524288	119194	1540816	2821891
1048576	308128	3004951	3426738
2097152	838936	3204392	2655642
4194304	1243438	8031806	5459939

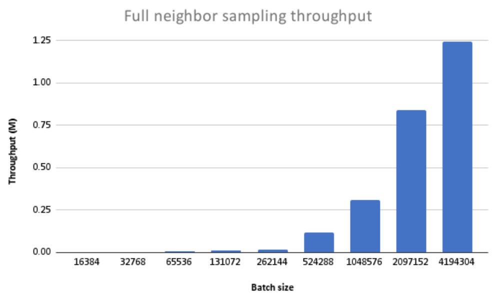
Table 5.2: Training throughput in terms of samples per second, w.r.t. batch size when using full neighbors set, limiting it to 50 neighbors and limiting to 25 neighbors during aggregation phase.

sampling, we expect the usage to be lower). It is worth noting that the memory usage increases after 524288 batch size, but oscillates for lower batch sizes. One possible hypothesis over the reason of the oscillation could be related to different allocation patterns used when allocating the data on GPU memory, however this requires further verification. Additionally, we can observe that even with the highest experimented batch size, the memory utilization has never exceeded the limits of our GPU memory, which could possibly allow us to train the model even on the entire graph at once.

As for our main performance measure, training throughput, we can see in Table 5.2 and visually in Figures 5.3a and 5.3b that it grows nearly linearly



(a)



(b)

Figure 5.3: (a) Training throughput w.r.t. batch size when setting a limit on maximum neighbors a node will aggregate information from. (b) Training throughput w.r.t batch size when all neighbors of a node are used to aggregate information.

with respect to the smallest batch size, in limited neighbor sampling scenarios. One interesting observation can regard the difference between throughputs for batch size of 524288 and 262144 when using full sampling. The former outperforms the latter by a factor of almost 6 times. No similar effect was observed for batch sizes bigger than 524288 for both full and limited sampling. Additionally, we can notice a throughout drop for batch size of 2097152, which could supposedly be fixed by better finetuning the dataloader's parameters.

Generally, internal observations have shown us that considering a batch size bigger than 524288 doesn't give the model enough training steps to converge to a desired accuracy value, without performing extensive parameter fine-tuning. In the subsequent experiments we thus fixed the batch size to 524288 and sampling limit to 25 and 50.

5.3 GPU scaling and math mode speedups

Having fixed the batch size to 524288 and sampling limits to 25 and 50, we have designed and executed our experiments to verify the following performance properties:

- **Math mode speedup**: speedup that can be obtained by using half-precision math mode (e.g. FP16, AMP) instead of single-precision (e.g. FP32) to perform the training.
- **Multi-GPU scaling**: how well does the model scale in a data parallel scenario where more than one GPU is used.
- **Hardware architecture speedup**: speedup that can be obtained by using a more powerful GPU architecture.

To verify these properties, we have performed 10 runs with a fixed seed on each configuration of the following parameter space:

1. **Sampling limit**: 25, 50.
2. **Math mode**: AMP, FP32/TF32.
3. **Number of GPUs**: 1, 8.
4. **Hardware architecture**: Volta, Ampere.

where for Volta runs we have used a DGX-1 machine which has 8 Volta V100 32GB GPUs. Ampere runs were done using a DGX-A100 machine, composed of 8 Ampere A100 80GB GPUs. It is worth mentioning that for Ampere, we have used TensorFloat32 format[2] for the single precision setups, which provides a speedup over the original FP32 format.

In Figure 5.4 we can see the throughput distributions for each configuration when using a Volta machine. We see from both the figure and Table 5.3 that for the sampling limit 25 configurations, AMP math mode brings a slowdown rather than a speedup. This could be attributed to the fact that AMP functionality is still in development in DGL[3], which might affect both message passing and kernels computation performance when working on float16 data type. Multi-GPU scaling on the other hand does occur for both AMP and

[2]https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/
[3]https://docs.dgl.ai/en/0.7.x/guide/mixed_precision.html

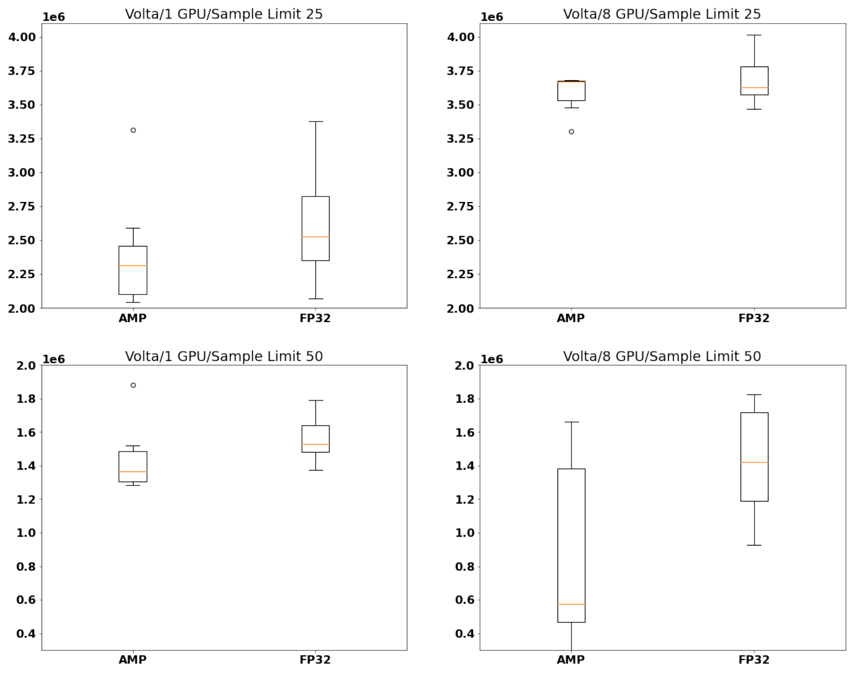


Figure 5.4: Training throughput distributions on DGX-1 (Volta) machine.

Sampling Limit	Precision	N. GPU	Throughput (mean)	Throughput (std)
25	AMP	1	2375711	359598
25	AMP	8	3591807	118668
25	FP32	1	2640575	406408
25	FP32	8	3676679	164569
50	AMP	1	1424820	172033
50	AMP	8	845969	510883
50	FP32	1	1562211	139461
50	FP32	8	1428676	313727

Table 5.3: Training throughput statistics for configurations on DGX-1 (Volta) machine.

Sampling Limit	Precision	N. GPU	Throughput (mean)	Throughput (std)
25	AMP	1	6440930	60346
25	AMP	8	5874435	223926
25	TF32	1	6883845	118461
25	TF32	8	6011209	592515
50	AMP	1	3449566	110830
50	AMP	8	2802328	142915
50	TF32	1	3694847	101338
50	TF32	8	3053406	242606

Table 5.4: Training throughput statistics for configurations on DGX-A100 (Ampere) machine.

FP32 scenarios when using sampling limit 25. In fact, we get a 1.51 scaling when using 8 GPUs AMP and 1.39 scaling when using 8 GPUs FP32, both of those scalings, however are poor scaling values. Using a sampling limit of 50 yields worse results in terms of scaling. For both AMP and FP32 precisions, the scaling is negative, meaning that using more GPUs for training actually deteriorates the training throughput. A possible root-cause of such low or even negative scaling can be reconducted to the throughputs when using a smaller batch-size. In fact, in our multi-GPU scenario, each GPU actually works with a smaller batch size (i.e. each GPU works on 65536 batch size, when the global batch size is 524288), on subgraphs with potentially different connectivity patterns, which as seen previously in Table 5.2, yields a significantly lower throughput, as well as introducing a synchronization overhead during the training.

As for a different hardware platform, in Figure 5.5 we can see training throughput distributions obtained on an Ampere machine. We see from the figure and the Table 5.4 that similarly to Volta, Ampere does not provide a speedup when using automatic mixed precision for both 25 and 50 sampling limits. Additionally, all the multi-GPU configurations bring a slowdown with respect to single GPU throughput. Both the previous performance concerns could be due to Ampere platform's operations not being properly optimized at lower level of DGL, given its novelty. One thing worth noting is however the

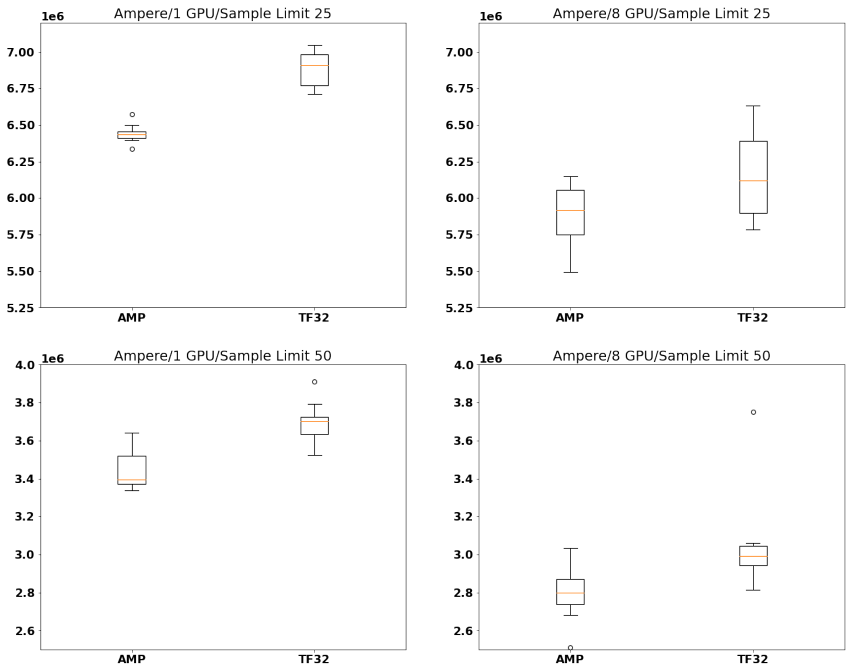


Figure 5.5: Training throughput distributions on DGX-A100 (Ampere) machine.

speedup given by Ampere with respect to Volta on every tested configuration. In fact, we can observe from Table 5.7 that Ampere beats Volta on every configuration, which is to be expected due to its introductions of TensorFloat32 precision format, as well as improved sparsity acceleration.

Lastly, we summarize the previous performance results in Tables 5.6, 5.5 and 5.7. Table 5.6 shows us results for the previously mentioned math mode speedups, which as already seen, do not occur, possibly due to both float16

Architecture	Sampling Limit	Precision	Multi-GPU Scaling
Volta	25	AMP	1.51
Volta	25	FP32	1.39
Volta	50	AMP	0.59
Volta	50	FP32	0.91
Ampere	25	AMP	0.91
Ampere	25	TF32	0.87
Ampere	50	AMP	0.81
Ampere	50	TF32	0.83

Table 5.5: Multi-GPU scaling values for the tested configurations.

Architecture	Sampling Limit	N. GPU	Precision Speedup
Volta	25	1	0.90
Volta	25	8	0.98
Volta	50	1	0.91
Volta	50	8	0.59
Ampere	25	1	0.94
Ampere	25	8	0.98
Ampere	50	1	0.93
Ampere	50	8	0.92

Table 5.6: Math mode speedup values for the tested configurations.

Sampling Limit	N. GPU	Precision	Platform Speedup
25	1	AMP	2.71
25	8	AMP	1.64
50	1	AMP	2.42
50	8	AMP	3.31
25	1	TF32/FP32	2.61
25	8	TF32/FP32	1.63
50	1	TF32/FP32	2.37
50	8	TF32/FP32	2.14

Table 5.7: Hardware platform speedup values for the tested configurations.

message passing and kernel calculation being still in development on DGL. Table 5.5 shows us the results for multi-GPU scaling, which are modest, presumably due to the fact that in our scenario, the local batch size on individual GPUs becomes too small to be possibly effective on the used hardware. Finally, Table 5.7 shows us speedups given by using a more powerful hardware platform, Ampere, which we observed always gave a speedup with respect to Volta, possibly due to its improved hardware capabilities such as better single precision math mode and improved sparsity acceleration.

Chapter 6

Conclusions

We have introduced recommendation systems and graph neural networks as two separate fields and later described how the two can be combined. We have seen that graph based recommendation approaches can naturally exploit the graph nature of recommendation data, and illustrated a pool of graph based models for recommendation that are available in the literature. Next, we have categorized the set of described models by characteristics that can have effect on their performance, namely: type of initial features they use, type of sampling they perform and their size in terms of parameters count. Finally, we have defined a simple model that mimicked the described models from a performance point of view, and used it to empirically study a set of relevant performance properties: math mode speedup, multi-GPU scaling and platform speedup.

In our study of the models available in the literature, we have observed some important points:

1. Most of the analyzed graph based recommendation models use the whole neighbors set instead of sampling a subset of K neighbors, which are later used to perform aggregation. Neighbor subset sampling is a common technique used to scale graph models, as it has a direct effect on both model and data loading performance. The latter was also empirically verified by us in Chapter 5.

2. Most of the analyzed models try to solve the recommendation task by using collaborative filtering approaches, which consists in using only user and item ids, despite having architectures that could easily include other types of data (e.g. by concatenation to node features), and training on datasets that actually provide that type of data. Using additional features other than ids can have an effect on model's performance, for example by including a fine-tuneable model branch that can encode image or textual data.

3. A huge portion of graph based recommendation model parameters are related to embedding layers, the parameter count from graph based layers is negligible in comparison. In fact, we have seen in Figure 4.21b that majority of parameters are composed of mainly embedding parameters, which means that on recommendation tasks, graph based models are not very different in terms of parameters behavior from their deep learning counter-parts, such as NCF.

From our performance experiments, using DGL, we have observed that:

1. Model training when using the automatic mixed precision math mode, does not give any performance benefits. It instead gives a slowdown, which could possibly be due to the fact that float16 message passing and kernel calculations, are still a feature in development in DGL, as stated in their documentation.

2. Model training on multi-GPUs, does not always scale well. When the scaling was positive, it was at most by a factor of 1.51 with respect to a single GPU. However, the majority of time, the scaling turned out to be negative, leading to a slowdown of the training. This could be possibly due to a too small batch size being used, resulting in a poor workload on individual GPUs, as well as the communication overhead induced by this batch size.

3. Changing hardware architecture to Ampere, which is a newer type of GPUs with better support for sparsity and tensor arithmetics, resulted in a performance boost over Volta on all the tested configurations, without requiring any code adaptations.

Next steps

We conclude this dissertation with suggestions on the possible future directions. First, it would be important to verify that the root cause of multi-GPU scaling problem is related to what we have described, namely, every GPU working on a small and inefficient batch size, introducing potential overheads rather than benefits. To do so, we suggest to try using a bigger batch size for every GPU and see how it affects the throughput. Next, we expect that once edge sampling on GPU becomes available as a feature of DGL, it will affect the performance, possibly the multi-GPU scaling, as each GPU would be able to sample its own portion of the graph, without requiring synchronization with the CPU. At the moment, edge sampling in DGL is only possible on CPU. Finally, it would be important to repeat the same set of experiments on a real model, once an implementation with all the before-mentioned features becomes available. This will allow us to verify how effectively our simple model can be used as a proxy to study the performance of more complicated models: given the same experimental settings, we do not expect the results to be very different across the two.

Bibliography

[1] Z. Abbassi, S. Amer-Yahia, L. V. Lakshmanan, S. Vassilvitskii, and C. Yu. Getting recommender systems to think outside the box. In *Proceedings of the Third ACM Conference on Recommender Systems*, RecSys '09, pages 285–288, New York, New York, USA. Association for Computing Machinery, 2009. ISBN: 9781605584355. DOI: `10.1145/1639714.1639769`. URL: `https://doi.org/10.1145/1639714.1639769`.

[2] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749, 2005. DOI: `10.1109/TKDE.2005.99`.

[3] S. Bell, Y. Liu, S. Alsheikh, Y. Tang, E. Pizzi, M. Henning, K. Singh, O. Parkhi, and F. Borisyuk. *Groknet: unified computer vision model trunk and embeddings for commerce*. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Association for Computing Machinery, New York, NY, USA, 2020, pages 2608–2616. ISBN: 9781450379984. URL: `https://doi.org/10.1145/3394486.3403311`.

[4] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi. Graph neural networks with convolutional arma filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*:1–1, 2021. ISSN: 1939-3539. DOI:

10.1109/tpami.2021.3054830. URL: http://dx.doi.org/10.1109/TPAMI.2021.3054830.

[5] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, October 2008. ISSN: 1742-5468. DOI: 10.1088/1742-5468/2008/10/p10008. URL: http://dx.doi.org/10.1088/1742-5468/2008/10/P10008.

[6] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, July 2017. ISSN: 1558-0792. DOI: 10.1109/msp.2017.2693418. URL: http://dx.doi.org/10.1109/MSP.2017.2693418.

[7] C. Cangea, P. Veličković, N. Jovanović, T. Kipf, and P. Liò. Towards sparse hierarchical graph classifiers, 2018. arXiv: 1811.01287 [stat.ML].

[8] L. Chen, L. Wu, R. Hong, K. Zhang, and M. Wang. Revisiting graph based collaborative filtering: a linear residual graph convolutional network approach, 2020. arXiv: 2001.10167 [cs.IR].

[9] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah. Wide & Deep Learning for Recommender Systems. *arXiv e-prints*:arXiv:1606.07792, arXiv:1606.07792, June 2016. arXiv: 1606.07792 [cs.LG].

[10] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh. Cluster-gcn. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, July 2019. DOI: 10.1145/3292500.3330925. URL: http://dx.doi.org/10.1145/3292500.3330925.

[11] A. Derrow-Pinion, J. She, D. Wong, O. Lange, T. Hester, L. Perez, M. Nunkesser, S. Lee, X. Guo, B. Wiltshire, and et al. Eta prediction with graph neural networks in google maps. *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, October 2021. DOI: `10.1145/3459637.3481916`. URL: `http://dx.doi.org/10.1145/3459637.3481916`.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: pre-training of deep bidirectional transformers for language understanding, 2019. arXiv: `1810.04805 [cs.CL]`.

[13] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The yahoo! music dataset and kdd-cup'11. In *Proceedings of the 2011 International Conference on KDD Cup 2011 - Volume 18*, KDDCUP'11, pages 3–18. JMLR.org, 2011.

[14] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin. Graph neural networks for social recommendation, 2019. arXiv: `1902.07243 [cs.IR]`.

[15] F. B. Fuchs, D. E. Worrall, V. Fischer, and M. Welling. Se(3)-transformers: 3d roto-translation equivariant attention networks, 2020. arXiv: `2006.10503 [cs.LG]`.

[16] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, December 1992. ISSN: 0001-0782. DOI: `10.1145/138859.138867`. URL: `https://doi.org/10.1145/138859.138867`.

[17] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He. Deepfm: a factorization-machine based neural network for ctr prediction, 2017. arXiv: `1703.04247 [cs.IR]`.

[18] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs, 2018. arXiv: `1706.02216 [cs.SI]`.

[19] R. He and J. McAuley. Ups and downs: modeling the visual evolution of fashion trends with one-class collaborative filtering. *Proceedings of the 25th International Conference on World Wide Web*, April 2016. DOI: `10.1145/2872427.2883037`. URL: `http://dx.doi.org/10.1145/2872427.2883037`.

[20] X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang. Lightgcn: simplifying and powering graph convolution network for recommendation, 2020. arXiv: `2002.02126 [cs.IR]`.

[21] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua. Neural collaborative filtering, 2017. arXiv: `1708.05031 [cs.IR]`.

[22] M. Jamali and M. Ester. A matrix factorization technique with trust propagation for recommendation in social networks. In *RecSys '10*, 2010.

[23] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken. Redundancy-free computation for graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pages 997–1005, Virtual Event, CA, USA. Association for Computing Machinery, 2020. ISBN: 9781450379984. DOI: `10.1145/3394486.3403142`. URL: `https://doi.org/10.1145/3394486.3403142`.

[24] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998. DOI: `10.1137/S1064827595287997`. eprint: `https://doi.org/10.1137/S1064827595287997`. URL: `https://doi.org/10.1137/S1064827595287997`.

[25] D. P. Kingma and M. Welling. Auto-encoding variational bayes, 2014. arXiv: `1312.6114 [stat.ML]`.

[26] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks, 2017. arXiv: `1609.02907 [cs.LG]`.

[27] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009. DOI: 10.1109/MC.2009.263.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[29] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks, 2017. arXiv: 1511.05493 [cs.LG].

[30] D. Liang, L. Charlin, J. McInerney, and D. M. Blei. Modeling user exposure in recommendation, 2016. arXiv: 1510.07025 [stat.ML].

[31] D. Liang, R. G. Krishnan, M. D. Hoffman, and T. Jebara. Variational autoencoders for collaborative filtering, 2018. arXiv: 1802.05814 [stat.ML].

[32] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: a robustly optimized bert pretraining approach, 2019. arXiv: 1907.11692 [cs.CL].

[33] Z. Liu, L. Meng, F. Jiang, J. Zhang, and P. S. Yu. Deoscillated graph collaborative filtering, 2021. arXiv: 2011.02100 [cs.IR].

[34] H. Ma, D. Zhou, C. Liu, M. R. Lyu, and I. King. Recommender systems with social regularization. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM ’11, pages 287–296, Hong Kong, China. Association for Computing Machinery, 2011. ISBN: 9781450304931. DOI: 10.1145/1935826.1935877. URL: https://doi.org/10.1145/1935826.1935877.

[35] J. McAuley, C. Targett, Q. Shi, and A. van den Hengel. Image-based recommendations on styles and substitutes, 2015. arXiv: 1506.04757 [cs.CV].

[36] F. Monti, M. M. Bronstein, and X. Bresson. Geometric matrix completion with recurrent multi-graph neural networks, 2017. arXiv: 1704.06803 [cs.LG].

[37] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.-H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models, 2021. arXiv: 2104.05158 [cs.DC].

[38] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems, 2019. arXiv: 1906.00091 [cs.IR].

[39] X. Ning and G. Karypis. Slim: sparse linear methods for top-n recommender systems. *2011 IEEE 11th International Conference on Data Mining*:497–506, 2011.

[40] P. Qi, X. Zhu, G. Zhou, Y. Zhang, Z. Wang, L. Ren, Y. Fan, and K. Gai. Search-based user interest modeling with lifelong sequential behavior data for click-through rate prediction, 2020. arXiv: `2006.05639 [cs.IR]`.

[41] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: unified, real-time object detection, 2016. arXiv: `1506.02640 [cs.CV]`.

[42] S. Rendle. Factorization machines. In *2010 IEEE International Conference on Data Mining*, pages 995–1000, 2010. DOI: `10.1109/ICDM.2010.127`.

[43] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme. Bpr: bayesian personalized ranking from implicit feedback. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pages 452–461, Montreal, Quebec, Canada. AUAI Press, 2009. ISBN: 9780974903958.

[44] O. Ronneberger, P. Fischer, and T. Brox. U-net: convolutional networks for biomedical image segmentation, 2015. arXiv: `1505.04597 [cs.CV]`.

[45] G. Shani, R. I. Brafman, and D. Heckerman. An mdp-based recommender system. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, UAI'02, pages 453–460, Alberta, Canada. Morgan Kaufmann Publishers Inc., 2002. ISBN: 1558608974.

[46] J. F. Silva, N. Moura Junior, and L. Caloba. Effects of data sparsity on recommender systems based on collaborative filtering. In pages 1–8, July 2018. DOI: `10.1109/IJCNN.2018.8489095`.

[47] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-J. (Hsu, and K. Wang. An overview of microsoft academic service (mas) and applications. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, pages 243–246, Florence, Italy. Association for

Computing Machinery, 2015. ISBN: 9781450334730. DOI: 10.1145/2740908.2742839. URL: https://doi.org/10.1145/2740908.2742839.

[48] H. O. Song, Y. Xiang, S. Jegelka, and S. Savarese. Deep metric learning via lifted structured feature embedding, 2015. arXiv: 1511.06452 [cs.CV].

[49] W. Song, Z. Xiao, Y. Wang, L. Charlin, M. Zhang, and J. Tang. Session-based social recommendation via dynamic graph attention networks. *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, January 2019. DOI: 10.1145/3289600.3290989. URL: http://dx.doi.org/10.1145/3289600.3290989.

[50] Y. Sun, D. Liang, X. Wang, and X. Tang. Deepid3: face recognition with very deep neural networks, 2015. arXiv: 1502.00873 [cs.CV].

[51] M. Tan and Q. V. Le. Efficientnet: rethinking model scaling for convolutional neural networks, 2020. arXiv: 1905.11946 [cs.LG].

[52] W. Torng and R. B. Altman. Graph convolutional neural networks for predicting drug-target interactions. *Journal of Chemical Information and Modeling*, 59(10):4131–4149, 2019. DOI: 10.1021/acs.jcim.9b00628. eprint: https://doi.org/10.1021/acs.jcim.9b00628. URL: https://doi.org/10.1021/acs.jcim.9b00628. PMID: 31580672.

[53] K. Tunyasuvunakool, J. Adler, Z. Wu, T. Green, M. Zielinski, A. Žídek, A. Bridgland, A. Cowie, C. Meyer, A. Laydon, S. Velankar, G. Kleywegt, A. Bateman, R. Evans, A. Pritzel, M. Figurnov, O. Ronneberger, R. Bates, S. Kohl, and D. Hassabis. Highly accurate protein structure prediction for the human proteome. *Nature*, 596:1–9, August 2021. DOI: 10.1038/s41586-021-03828-1.

[54] R. van den Berg, T. N. Kipf, and M. Welling. Graph convolutional matrix completion, 2017. arXiv: 1706.02263 [stat.ML].

[55] A. van den Oord, S. Dieleman, and B. Schrauwen. Deep content-based music recommendation. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL: https://proceedings.neurips.cc/paper/2013/file/b3ba8f1bee1238a2f37603d90b58898d-Paper.pdf.

[56] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2017. arXiv: 1706.03762 [cs.CL].

[57] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks, 2018. arXiv: 1710.10903 [stat.ML].

[58] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. Deep graph library: a graph-centric, highly-performant package for graph neural networks, 2020. arXiv: 1909.01315 [cs.LG].

[59] W. Wang, W. Zhang, S. Liu, Q. Liu, B. Zhang, L. Lin, and H. Zha. Beyond clicks: modeling multi-relational item graph for session-based target behavior prediction, 2021. arXiv: 2002.07993 [cs.IR].

[60] X. Wang, X. He, M. Wang, F. Feng, and T.-S. Chua. Neural graph collaborative filtering. *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, July 2019. DOI: 10.1145/3331184.3331267. URL: http://dx.doi.org/10.1145/3331184.3331267.

[61] X. Wang, H. Jin, A. Zhang, X. He, T. Xu, and T.-S. Chua. Disentangled graph collaborative filtering. *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, July 2020. DOI: 10.1145/3397271.3401137. URL: http://dx.doi.org/10.1145/3397271.3401137.

[62] X. Wang, R. Wang, C. Shi, G. Song, and Q. Li. Multi-component graph convolutional collaborative filtering, 2019. arXiv: 1911.10699 [cs.LG].

[63] F. Wu, T. Zhang, A. H. de Souza Jr. au2, C. Fifty, T. Yu, and K. Q. Weinberger. Simplifying graph convolutional networks, 2019. arXiv: 1902.07153 [cs.LG].

[64] L. Wu, J. Li, P. Sun, R. Hong, Y. Ge, and M. Wang. Diffnet++: a neural influence and interest diffusion network for social recommendation, 2021. arXiv: 2002.00844 [cs.SI].

[65] L. Wu, P. Sun, Y. Fu, R. Hong, X. Wang, and M. Wang. A neural influence diffusion model for social recommendation, 2019. arXiv: 1904.10322 [cs.IR].

[66] Q. Wu, H. Zhang, X. Gao, P. He, P. Weng, H. Gao, and G. Chen. Dual graph attention networks for deep latent representation of multifaceted social effects in recommender systems. *The World Wide Web Conference*, May 2019. DOI: 10.1145/3308558.3313442. URL: http://dx.doi.org/10.1145/3308558.3313442.

[67] S. Wu, F. Sun, W. Zhang, and B. Cui. Graph neural networks in recommender systems: a survey, 2021. arXiv: 2011.02260 [cs.IR].

[68] S. Wu, Y. Tang, Y. Zhu, L. Wang, X. Xie, and T. Tan. Session-based recommendation with graph neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:346–353, July 2019. ISSN: 2159-5399. DOI: 10.1609/aaai.v33i01.3301346. URL: http://dx.doi.org/10.1609/aaai.v33i01.3301346.

[69] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling, 2019. arXiv: 1806.08804 [cs.LG].

[70] F. Yu, Y. Zhu, Q. Liu, S. Wu, L. Wang, and T. Tan. Tagnn: target attentive graph neural networks for session-based recommendation. *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, July 2020. DOI: `10.1145/3397271.3401319`. URL: `http://dx.doi.org/10.1145/3397271.3401319`.

[71] J. Zhang, X. Shi, S. Zhao, and I. King. Star-gcn: stacked and reconstructed graph convolutional networks for recommender systems, 2019. arXiv: `1905.13129 [cs.IR]`.

[72] L. Zheng, C.-T. Lu, F. Jiang, J. Zhang, and P. S. Yu. Spectral collaborative filtering. *Proceedings of the 12th ACM Conference on Recommender Systems*, September 2018. DOI: `10.1145/3240323.3240343`. URL: `http://dx.doi.org/10.1145/3240323.3240343`.

[73] G. Zhou, N. Mou, Y. Fan, Q. Pi, W. Bian, C. Zhou, X. Zhu, and K. Gai. Deep interest evolution network for click-through rate prediction, 2018. arXiv: `1809.03672 [stat.ML]`.

[74] G. Zhou, C. Song, X. Zhu, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai. Deep interest network for click-through rate prediction, 2018. arXiv: `1706.06978 [stat.ML]`.

Acknowledgements

I would first like to thank everyone who during my academic career has shaped my views and motivated me to pursue a career in Artificial Intelligence. Since my first day as a Computer Science Bachelor's degree student I was dreaming to specialize in AI in the future. It was a joy for me when University of Bologna has opened a specialized Master's degree course in Artificial Intelligence, allowing me to pursue my goals. I would like to collectively thank all of the professors from my Master's degree for providing me with enough technical knowledge to approach any problem with confidence.

I would like to thank my thesis supervisor, prof. Paolo Torroni for his organizational work and willingness to receive student feedback, which has helped to greatly increase the quality of the Artificial Intelligence Master's degree course. Additionally, I would like to thank him in the role of my professor of Natural Language Processing course, which provided me with a solid practical and theoretical knowledge of the field. Finally, I would like to thank him in the role of my thesis supervisor, for supporting my work and helping me to improve its overall quality and scope.

I would like to thank Piotr Bigaj, my manager and internship supervisor at NVIDIA, for his support and understanding throughout the entire duration of my internship. Additionally, I would like to thank him for introducing me to the field of recommendation systems, as well as optimizing deep learning methods on powerful industrial hardware. I would also like to collectively thank my collegues from NVIDIA for their overall support and discussions, helping me to improve my knownledge of deep learning framework internals, nuances and problems, as well as helping me to develop methodologies on how to deal with them.

I would like to thank my parents, Olga and Alessandro, for never stopping to support me throughout the whole duration of my university studies and for welcoming my choice of relocating to Warsaw, Poland to perform my internship.

I would like to thank my girlfriend Serafima, for being very close, supportive and empathetic with me at the time of writing this thesis. I still remember the period where I would reply her “Thesis..Chapter 4” every time she would ask me what was on my mind.

Finally, I would like to thank all the class mates from the university with whom I’ve interacted or done team projects with, during the duration of the Master’s degree.