Part IV
Broad and Emerging Applications with Graph Neural Networks

Chapter 19
Graph Neural Networks in Modern Recommender Systems

Yunfei Chu, Jiangchao Yao, Chang Zhou and Hongxia Yang

Abstract Graph is an expressive and powerful data structure that is widely applicable, due to its flexibility and effectiveness in modeling and representing graph structure data. It has been more and more popular in various fields, including biology, finance, transportation, social network, among many others. Recommender system, one of the most successful commercial applications of the artificial intelligence, whose user-item interactions can naturally fit into graph structure data, also receives much attention in applying graph neural networks (GNNs). We first summarize the most recent advancements of GNNs, especially in the recommender systems. Then we share our two case studies, dynamic GNN learning and device-cloud collaborative Learning for GNNs. We finalize with discussions regarding the future directions of GNNs in practice.

19.1 Graph Neural Networks for Recommender System in Practice

19.1.1 Introduction

The Introduction of GNNs Graph has a long history originated from the Seven Bridges of Königsberg problem in 1736 (Biggs et al, 1986). It is flexible to model

Yunfei Chu,
DAMO Academy, Alibaba Group, e-mail: fay.cyf@alibaba-inc.com

Jiangchao Yao
DAMO Academy, Alibaba Group, e-mail: jiangchao.yjc@alibaba-inc.com

Chang Zhou
DAMO Academy, Alibaba Group, e-mail: ericzhou.zc@alibaba-inc.com

Hongxia Yang
DAMO Academy, Alibaba Group, e-mail: yang.yhx@alibaba-inc.com

complex relationships among individuals, which makes it a ubiquitous data structure widely applied in numerous fields, *e.g.*, biology, finance, transportation, social network, recommender systems.

Despite there are traditional topics of extracting deterministic information in graph theory like shortest path, connected components, local clustering, graph isomorphism, and *etc.*, machine learning applications for graph data focus more on predicting the missing parts or future dynamics. Among these applications, the most typical research problems studied in recent year, are predicting whether there exists or will emerge an edge between two nodes (link prediction), and inferring node-level or graph-level labels (node/graph classification).

The recent progress in deep learning leads to a booming learning paradigm called representation learning, which also becomes the de facto standard in solving graph machine learning problems. The idea of graph representation learning is to encode graph primitives as real-valued vectors in the same metric space, which are then involved in downstream applications. The encoder takes as input the original graph such as node attributes vector and graph adjacency matrix in an end-to-end fashion, rather than traditional methods that require extracting heuristic features such as betweenness centrality, pagerank value, number of closed triangles.

Next, we summarize recent graph node representation techniques in a unified framework and focus only on the link prediction task. We illustrate several representative approaches in recent literature from a node-centric perspective, since the node-centric view can naturally fit into scalable message passing implementations that are originally popular in graph mining community (Malewicz et al, 2010; Y.Low et al, 2012) and then borrowed to GNNs community (Wang et al, 2019f; Zhu et al, 2019c).

For a graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ with adjacency matrix A, a standard graph neural network model has the following components.

- An ego-network extractor EGO that extracts a local subgraph around the node v. This local subgraph is also referred to as the receptive field of v which is then used by the node encoder.
- An encoder ENC that maps each node $v \in \mathscr{V}$ into a vector in a metric space R^d. The encoder takes as input the ego-network of v, as well as any node/edge representation in $EGO(v)$. A similarity function is defined on R^d to measure how close two nodes appear to be.
- A learning objective $\mathscr{L}$. We do not discuss node classification here and only focus on unsupervised node representation learning. The objective can be reconstructing the adjacency matrix A, transformations of A, or any sampled form of A and its transformations.

Random Walk-style

Early graph representation learning approaches (Perozzi et al, 2014; Tang et al, 2015b; Cao et al, 2015; Zhou et al, 2017; Ou et al, 2016; Grover and Leskovec, 2016) in deep learning era are inspired by word2vec (Mikolov et al, 2013b), an efficient word embedding method in natural language processing community. These

methods do not need any neighborhood for encoding, where EGO plays as an identity mapping. The encoder ENC takes as the node id in the graph and assigns a trainable vector to each node.

The very different part of these methods is the learning objective. Approaches like Deepwalk, LINE, Node2vec use different random walk strategies to create positive node pairs (u,v) as the training example, and estimate the probability of visiting v given u, $p(v|u)$, as a multinomial distribution,

$$p(v|u) = \frac{\exp(sim(u,v))}{\sum_{v'} \exp(sim(u,v'))},$$

where sim is a similarity function. They exploit an approximated Noise Constrained Estimation (NCE) loss (Gutmann and Hyvärinen, 2010), known as skip gram with negative sampling originated in word2vec as the following, to reduce the high computation cost,

$$\log \sigma(sim(u,v)) + k\mathbb{E}_{v' \sim q_{neg}} \log(1 - \sigma(sim(u,v'))).$$

q_{neg} is a proposed negative distribution, which impacts the variation of the optimization target (Yang et al, 2020d). Note that this formula can be also approximated with sampled softmax (Bengio and Senécal, 2008; Jean et al, 2014), which in our experience performs better in top-k recommendation tasks as the node number becomes extremely large (Zhou et al, 2020a).

These learning objectives have connections with traditional node proximity measurements in graph mining community. GraRep (Cao et al, 2015), APP (Zhou et al, 2017) borrows the idea from (Levy and Goldberg, 2014) and point out these random walk based method are equivalent to preserving their corresponding transformations of the adjacency matrix A, such as personalized pagerank.

Matrix Factorization-style

HOPE (Ou et al, 2016) provides a generalized matrix form of other types of node proximity measurement, *e.g.*, katz, adamic-adar, and adopts matrix factorization to learn embedding that preserve these proximity. NetMF (Qiu et al, 2018) unifies several classic graph embedding methods in the framework of matrix factorization, provides connections between the deepwalk-like approaches and the theory of graph Laplacian.

GNN-style

Graph neural network (Kipf and Welling, 2017b; Scarselli et al, 2008) provides an end-to-end semi-supervised learning paradigm that was previously modeled via label propagations. It can also be used to learn node representations in an unsupervised manner like the above graph embedding methods. GNN-like approaches for unsupervised learning, compared to deepwalk-like methods, are more powerful in capturing local structural, *e.g.*, have at most the power of WL-test (**?**). The

downstream link prediction task that requires local-structural aware representation or cooperation with node features may benefit more from GNN-style approaches.

The *EGO* operator collects and constructs the receptive field of each node. For GCN (Kipf and Welling, 2017b), a full k-layer neighborhood is required for each node, making it hard to work for large graphs which usually follow power-law degree distribution. GraphSage (Hamilton et al, 2017b) instead samples a fixed-size neighborhood in each layer, mitigates this problem and can scale to large graphs. LCGNN (Qiu et al, 2021) samples a local cluster around each node by short random-walks with theoretical guarantee.

Then different kinds of Aggregation functions are proposed within this receptive field. GraphSage investigates several neighborhood aggregation alternatives, including mean/max pooling, LSTM. GAT (Veličković et al, 2018) utilizes self-attention to perform the aggregation, which shows stable and superior performance in many graph benchmarks. GIN (Xu et al, 2019d) has a slightly different aggregation function, whose discriminative/representational power is proved to be equal to the power of the WL test. As link prediction task may also consider structural similarity between two nodes besides their distance, this local structural preserving method may achieve good performance for networks that have obvious local structural patterns.

The learning objectives of GNN-style approaches are similar with those in random walk style ones.

Introduction of Modern Recommender System

Recommender system, one of the most successful commercial applications of the artificial intelligence, whose user-item interactions can naturally fit into graph structure data, also receives much attention to applying GNNs. We now give a brief introduction about the problem settings, the classic methods in recommender systems.

The user-item relationships are the most typical form of recommender systems, *e.g.*, news recommendation, e-commerce recommendation, video recommendation. Although recommender systems are eventually optimizing for a complex ecosystem of multi-sided participants (Abdollahpouri et al, 2020), i.e., the users, the platform and the content provider, we only focus on how the platform will maximize the user-side utility in this chapter.

In a user-item recommender system $\mathscr{S}$ with recommender algorithm $\mathscr{A}$, $\mathscr{U}$ is the user set and $\mathscr{I}$ is the item set. At timestamp t, a user $u \in \mathscr{U}$ visits $\mathscr{S}$, a list of items $\mathscr{I}_{u,t}$ is produced by $\mathscr{A}$. u takes positive actions, *e.g.*, click, buy, play, on parts of the items in $\mathscr{I}_{u,t}$, referred to as $\mathscr{I}^+_{u,t}$, while performing the corresponding negative actions on the others, *e.g.*, not click, not buy, not play, referred to as $\mathscr{I}^-_{u,t}$.

The basic data collected from an industrial recommender system, can be described as

$$\mathscr{D}_{\mathscr{S},\mathscr{A}} = \{(t, \mathscr{I}^+_{u,t}, \mathscr{I}^-_{u,t}) | u \in \mathscr{U}, t\}. \tag{19.1}$$

The short-term objective [1] of an algorithm in modern recommender systems, can be summarized as

$$\mathscr{A} = \arg\max_{\mathscr{A}} \sum_{u,t} Utility(\mathscr{I}^+_{u,t}), \tag{19.2}$$

in which the $Utility$ function could be considered as maximizing *click through rate*, *GMV*, or a mixture of multiple objectives (Ribeiro et al, 2014; McNee et al, 2006).

A modern commercial recommender system, especially for those with over millions of end-users and items, has adopted a multi-stage modeling pipeline as the tradeoff between the business goals and the efficiency given the constraints of limited computing resources. Different stages have different simplifications of the data organization and objectives, which many research papers do not put in a clear way.

In the following, we first review several simplifications of the industrial recommendation problem setting, that are clean enough for the research community. Then we describe the multi-stage pipeline and the problem in each stage, review classic methods to handle the problem and revisit how GNNs are applied in existing methods, trying to give an objective view about these methods.

Simplifications of the collected data.

- *Impression bias.* The user feedback data generated under algorithm $\mathscr{A}$, has a bias towards estimating the oracle user preference. This critical and unique problem for recommender system, is usually not considered, especially for the early works.
- *Negative feedback.* $|\mathscr{L}^-_{u,t}|$, the number of negative behaviors in one display, is orders of magnitude larger than $|\mathscr{L}^+_{u,t}|$, and very few dataset has collected negative feedback. Most of the well-known papers in the research community ignore those true negative user feedback, instead, they simulate negative feedback by sampling from a proposal distribution, which is not the ground truth and the metrics designed over the simulated feedback may not reveal true performance.
- *Temporal information.* Early studies prefer a static view of recommendation, which eliminates the temporal information of t in the user behavior sequences.

Multi-stage model pipeline in modern recommender systems.

- Retrieval Phase. This phase is also referred to as candidate generation or recall phase. It narrows down the collection of relevant items from billions to hundreds via efficient similarity-based learning, indexing, and searching. To prevent from sticking into dead loops caused by fitting the exposure distribution, retrieval phase has to independently provide sufficient diversity for different downstream purposes or strategies, while retaining the accuracy. As the candidate set is in extremely large size, approaches in the recall phase are usually in the form of point-wise modeling that is simple to build sophisticated index and perform

[1] We indicate the short-term objective as the objective in the sense of each request response. Here we do not consider further impacts on the ecosystem brought by an algorithm.

Table 19.1: Data simplifications in different settings

Setting / Phase in Pipeline	Data Simplification
Matrix Completion / Retrieval Phase	$\mathscr{D}_{\mathscr{S}} = \{\mathscr{L}_u^+ \vert u \in \mathscr{U}\}$
Click Through Rate Prediction / Rank Phase	$\mathscr{D}_{\mathscr{S}} = \{(\mathscr{L}_u^+, \mathscr{L}_u^-) \vert u \in \mathscr{U}\}$
Sequential Recommendation / Retrieval Phase	$\mathscr{D}_{\mathscr{S}} = \{(t, \mathscr{L}_{u,t}^+) \vert u \in \mathscr{U}, t\}$

efficient retrieval. The most widely used measurement for this phase is the top-k hit ratio.

- Rank Phase. The problem space is quite different from those in the retrieval phase, since rank phase needs to give precise comparison within a much smaller subspace, instead of recalling as many as good items from the entire item candidates set. Restricted to a small number of candidates, it is capable of exploiting more complex methods over the user-item interaction in acceptable response time.
- Re-rank Phase. Considering the effects studied in the discrete choice model (Train, 1986), the relationships among the displayed items may have significant impacts on the user behavior. This poses opportunities to consider from the combinational optimization perspective, i.e., how to chose a combination of the subset which maximizes the whole utilities of the recommendation list.

The above stages can be adjusted according to different characteristics of the recommendation scenario. For example, if the candidate set is at hundreds or thousands, recall phase is not necessarily required as the computation power is usually enough to cover such rank-all operation at once. The re-rank phase is also not necessary if the item number per request is few.

We summarize in Table 19.1 the different data simplifications made in different problem settings with their corresponding pipeline stages.

19.1.2 Classic Approaches to Predict User-Item Preference

The fundamental ability required by Recommender System is to predict the possibility that a user will take actions on a specific displayed item, which we refer to as the point-wise preference estimation, $p(item|user)$. Now we review several classic approaches in dealing with the cleanest setting of Matrix Completion in Table 19.1.

The user-item iteraction matrix perspective of data organization $\mathscr{D}_{\mathscr{S}} = \{\mathscr{L}_u^+ \vert u \in \mathscr{U}\}$ is $M = \{M_{u,i} \vert u \in \mathscr{U}, i \in \mathscr{I}\}$, where each row $M_u = \mathscr{L}_u^+$. The famous Collaborative Filtering methods in recommendation can be categorized into neighborhood-based one and model-based one.

Neighborhood-based Approaches

Item-based collaborative filtering first identifies a set of similar items for each of the items that the user has clicked/purchased/rated, and then recommends top-N items by aggregating the similarities. User-based CF, on the other hand, identifies similar users and then performs aggregation on their clicked items.

The key part in Neighborhood-based Approaches is the definition of the similarity metric. Take item-based CF as an example, top-k heuristic approaches calculate item-item similarity from the user-item interaction matrix M, *e.g.*, pearson correlation, cosine similarity. Storing $|\mathscr{I}|x|\mathscr{I}|$ similarity score pairs is intractable. Instead, to help produce a top-k recommendation list efficiently, neighborhood-based k-nearest-neighbor CF usually memorizes top few similar items for each item, resulting in a sparse similarity matrix C. Despite the heuristics, SLIM (Ning and Karypis, 2011) learns such sparse similarity by reconstructing M via MC with zero diagonal and sparse constraints in C.

One draw back of storing only the sparse similarity is that, it cannot identify less-similar relationships which restricts its downstream applications.

Model-based Approaches

Model-based methods learn similarity functions between user and item by optimizing an objective function. Matrix Factorization, the prior of which is that the user-behavior matrix is low-rank, i.e., all users' tastes can be described by linear combinations of a few style latent factors. The prediction for a user's preference on an item can be calculated as the dot product of the corresponding user and item factor.

19.1.3 Item Recommendation in user-item Recommender Systems: a Bipartite Graph Perspective

The matrix completion setting also has an equivalent form in bipartite graph,

$$\mathscr{G} = (\mathscr{V}, \mathscr{E}), \tag{19.3}$$

where $\mathscr{V} = \mathscr{U} \cup \mathscr{I}$, *i.e.,* the union of the user set $\mathscr{U}$ and the item set $\mathscr{I}$, and $\mathscr{E} = \{(u,i)|i \in \mathscr{I}_u^+,\ u \in \mathscr{U}\}$, *i.e.,* the collection of the edges between u and his/her clicked i. Then the point-wise user-item preference estimation can be viewed as a link prediction task in this user-item interaction bipartite graph.

Heuristic graph mining approaches, which fall into the category of neighborhood-based CF, are widely used in the retrieval phase. We can calculate user-item similarity by performing graph mining tasks like Common Neighbors, Adar (Adamic and Adar, 2003), Katz (Katz, 1953), Personalized PageRank (Haveliwala, 2002), over the original bipartite graph, or calculate item-item similarity on its induced item-item correlation graph (Zhou et al, 2017; Wang et al, 2018b) which are then used in the final user preference aggregation.

Graph embedding techniques for industrial recommender system are first explored in (Zhou et al, 2017) and its successor with side information support (Wang et al, 2018b). They construct an item correlation graph of billions of edges from user-item click sequences organized by sessions. Then a deepwalk-style graph embedding method is applied to calculate the item representations, which then provides item-item similarities in the retrieval phase. Though it's shown in (Zhou et al, 2017) that embedding based method has advantage in scenarios where the top-k heuristics cannot provide any item-pair similarity, it's still debatable whether the similarity given by graph embedding methods can outperform carefully designed heuristic ones when all the top-k similar item can be retrieved.

We also note that, graph embedding techniques can be regarded as matrix factorization for a transformation of the graph adjacency matrix A, as discussed in earlier sections. That means, theoretically the difference between graph embedding techniques and the basic matrix factorization are their priors, i.e., what matrix is assumed to be the best to factorize. Factorization of the transformations of A indicates to fit an evolved system in the future while traditional MF methods are factorizing the current static system.

Graph neural networks for industrial recommender system are first studied in (Ying et al, 2018b), whose backend model is a variant of GraphSage. PinSage computes the L1 normalized visit counts of nodes during random walks started from a given node v, and the top-k counted nodes are regarded as v's receptive field. Weighted aggregation is performed among the nodes according to their normalized counts. As GraphSage-like approaches do not suffer from too large neighborhood, PinSage is scalable to web-scale recommender system with millions of users and items. It adopts a triplet loss, instead of NCE-variants that are usually used in other papers.

We want to discuss more about the choice of negative examples in representation learning based recommender models, including GNNs, in the retrieval phase. As retrieval phase aims to retrieve the k most relevant items from the entire item space, it's crucial to keep an item's global position far from *all* irrelevant items. In an industrial system with an extremely large candidate set, we find the performance of any representation-based model very sensitive to the choice of negative samples and the loss function. Though there seems a trend in mixing all kinds of hand-crafted hard examples (Ying et al, 2018b; Huang et al, 2020b; Grbovic and Cheng, 2018) in binary cross entropy loss or triplet loss, unfortunately, it has even no theoretical support that can lead us to the right direction. In practice, we find it a good choice to apply sampled softmax (Jean et al, 2014; Bengio and Senécal, 2008), InfoNCE (Zhou et al, 2020a) in the retrieval phase with an extremely large candidate set, where the latter has also an effect of debiasing.

GNNs are a useful tool to incorporate with relational features of user and item. KGCN (Wang et al, 2019e) enhances the item representation by performing aggregations among its corresponding entity neighborhood in a knowledge graph. KGNN-LS (Wang et al, 2019c) further poses a label smoothness assumption, which posits that similar items in the knowledge graph are likely to have similar user preference. It adds a regularization term to help learn such a personalized weighted

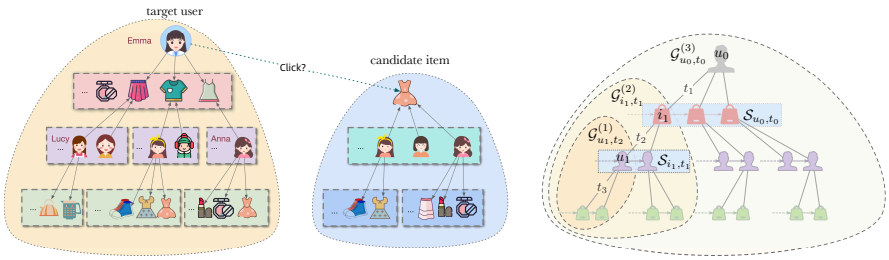
knowledge graph. KGAT (Wang et al, 2019j) shares a generally similar idea with KGCN. The only main difference is an auxiliary loss for knowledge graph reconstruction.

Despite there are many more paper discussing about how to fuse external knowledge, relationships of other entities, which all argue it's beneficial for downstream recommendation tasks, one should seriously consider whether its system needs such external knowledge or it will introduce more noises than benefits.

19.2 Case Study 1: Dynamic Graph Neural Networks Learning

19.2.1 Dynamic Sequential Graph

In a recommender, we can obtain a list of user-item interaction tuples $\mathscr{E} = \{(u,i,t)\}$ observed in a time window, where the user $u \in \mathscr{U}$ interacts with an item $i \in \mathscr{I}$ associated with an timestamp $t \in \mathbb{R}^+$. For a user $u \in \mathscr{U}$ (or an item $i \in \mathscr{I}$) at time t, we define the 1-depth dynamic sequential subgraph of user u (or item i) at time t as a set of interactions of user u (or item i) before time t in chronological order, denoted by $\mathscr{G}_{u,t}^{(1)} = \{(u,i,\tau)|\tau < t, (u,i,\tau) \in \mathscr{E}\}$ (or $\mathscr{G}_{i,t}^{(1)} = \{(u,i,\tau)|\tau < t, (u,i,\tau) \in \mathscr{E}\}$). Given the k-depth dynamic sequential subgraphs $\mathscr{G}_{i,t}^{(k)}$ for $i \in \mathscr{I}$ (or $\mathscr{G}_{u,t}^{(k)}$ for $u \in \mathscr{U}$), we define the (k+1)-depth dynamic sequential subgraph of user u (or item i) at time t as a set of k-depth dynamic sequential subgraphs that user u (or item i) interacts in chronological order with its 1-depth dynamic sequential subgraphs, $\mathscr{G}_{u,t}^{(k+1)} = \{\mathscr{G}_{i,\tau}^{(k)}|\tau < t, (u,i,\tau) \in \mathscr{E}\} \cup \mathscr{G}_{u,t}^{(1)}$ (or $\mathscr{G}_{i,t}^{(k+1)} = \{\mathscr{G}_{u,\tau}^{(k)}|\tau < t, (u,i,\tau) \in \mathscr{E}\} \cup \mathscr{G}_{i,t}^{(1)}$). The illustration of DSG is shown in Figure 19.1. We define the historical behavior sequence of user u (or item i) at time t as a sequence of interacted items (or users) in chronological order, denoted by $\mathscr{S}_{u,t} = \{(i,\tau)|\tau < t, (u,i,\tau) \in \mathscr{E}\}$ (or $\mathscr{S}_{i,t} = \{(u,\tau)|\tau < t, (u,i,\tau) \in \mathscr{E}\}$).



(a) Dynamic sequential graphs in recommendation. (b) An example of a user's 3-depth DSG.

Fig. 19.1: Illustration of Dynamic Sequential Graph. DSG is a heterogeneous time-evolving dynamic graph combining the high-hop connectivity in graphs and the temporal dependency in sequences. DSG is constructed from bottom to top recursively.

19.2.2 DSGL: Dynamic Sequential Graph Learning

19.2.2.1 Overview

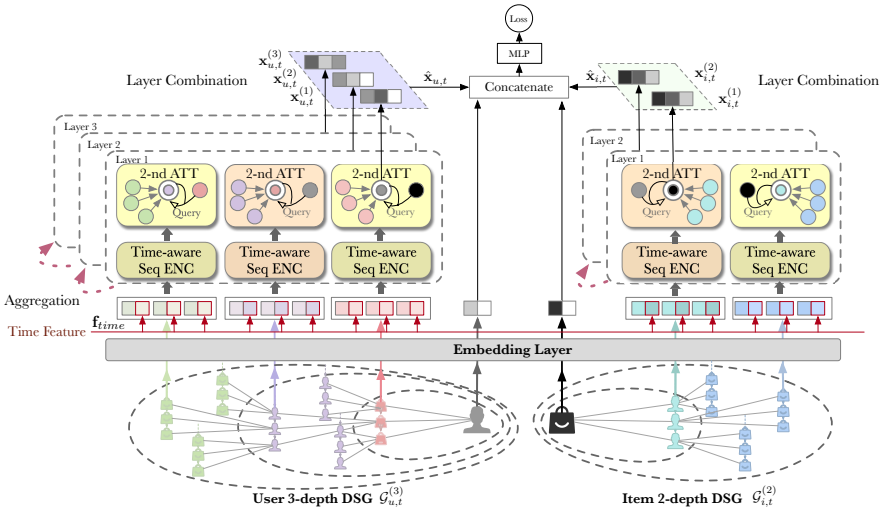


Fig. 19.2: Framework of the proposed DSGL method. DSGL constructs DSGs for the target user u (left) and the candidate item i (right) respectively. Their representations are refined with multiple aggregation layers, each of which consists of a time-aware sequence encoding layer and a second-order graph attention layer. DSGL gets the final representations via layer combination followed by an MLP-based prediction layer. Modules of the same color share the same set of parameters.

Based on the constructed user-item interaction DSG, we propose the edge learning model named Dynamic Sequential Graph Learning (DSGL), as illustrated in Figure 19.2. The basic idea of DSGL is to perform graph convolution iteratively on the DSGs for the target user and the candidate item on their corresponding devices, by aggregating the embeddings of neighbors as the new representation of a target node. The aggregator consists of two parts: (1) the time-aware sequence encoding that encodes the behavior sequence with time information and temporal dependency captured; and (2) the second-order graph attention that activates the related behavior in the sequence to eliminate noisy information. Besides the above two components, we also propose an embedding layer that initializes user, item, and time embeddings, a layer combination module that combines the embeddings of multiple layers to achieve final representations, and a prediction layer that outputs the prediction score.

19.2.2.2 Embedding Layer

There are four groups of inputs in the proposed DSGL: the target user u, the candidate item i, the k-depth DSGs of the target user $\mathscr{G}_{u,t}^{k}$ and (k-1)-depth DSGs of the candidate item $\mathscr{G}_{i,t}^{k-1}$. For each field of discrete features, such as age, gender,category, brand, and ID, we represent it as an embedding matrix. By concatenating all fields of features, we have the node feature of items, denoted by $\mathbf{f}_{item} \in \mathbb{R}^{d_i}$. Similarly, $\mathbf{f}_{user} \in \mathbb{R}^{d_u}$ represents the concatenated embedding vectors of fields in the category of user. As for the interaction timestamp in DSG, we compute the time intervals between the interaction time and its parent interaction time as time decays. Given a historical behavior sequence $\mathscr{S}_{u,t}$ of user u at the timestamp t, each interaction $(u,i,\tau) \in \mathscr{S}_{u,t}$ corresponds to a time decay $\Delta_{(u,i,\tau)} = t - \tau$. Following (Li et al, 2020g), we transform the continuous time decay values to discrete features by mapping them to a series of buckets with the ranges $[b^0,b^1),[b^1,b^2),\ldots,[b^l,b^{l+1})$, where the base b is a hyper-parameter. Then by performing the embedding lookup operation, the time decay embedding can be obtained, denoted by $\mathbf{f}_{time} \in \mathbb{R}^{d_t}$.

19.2.2.3 Time-Aware Sequence Encoding

The nodes at each layer of DSGs are in time order, which reflects the time-varying preference of users as well as the popularity evolution of items. Thus we perform sequence modeling as a part of GNN to capture the dynamics of the interaction sequences. We design a *time-aware sequential encoder* to utilize the time information explicitly. For each interaction (u,i,t), we have the historical behavior sequence $\mathscr{S}_{u,t}$ of user u and $\mathscr{S}_{i,t}$ of item i. For sequence $\mathscr{S}_{u,t}$, by feeding each interacted item along with the time decay in the sequence into the embedding layer, the behavior embedding sequence is formed with the combined feature sequence, as $\{\mathbf{e}_{i,\tau}|(i,\tau) \in \mathscr{S}_{u,t}\}$, where $\mathbf{e}_{i,\tau} = [\mathbf{f}_{item_i};\mathbf{f}_{time_\tau}] \in \mathbb{R}^{d_i+d_t}$ is the embedding of item i in the sequence. Similarly, for sequence $\mathscr{S}_{i,t}$, we have the embedding sequence as $\{\mathbf{e}_{u,\tau}|(u,\tau) \in \mathscr{S}_{i,t}\}$, where $\mathbf{e}_{u,\tau} = [\mathbf{f}_{user_u};\mathbf{f}_{time_\tau}] \in \mathbb{R}^{d_u+d_t}$. We take the obtained embedding as the zero-layer of inputs in the time-aware sequence encoder, i.e., $\mathbf{x}_{u,t}^{(0)} = \mathbf{e}_{u,t}$ and $\mathbf{x}_{i,t}^{(0)} = \mathbf{e}_{i,t}$. For ease of notation, we will drop the superscript in the rest of the following two subsections.

In the time-aware sequence encoding, we infer the hidden state of each node in the behavior sequence step by step in a RNN-based manner. Given the behavior sequences $\mathscr{S}_{u,t}$ and $\mathscr{S}_{i,t}$, we represent j-th item's hidden states and inputs in the sequence $\mathscr{S}_{u,t}$ as $\mathbf{h}_{item_j}$ and $\mathbf{x}_{item_j}$, and j-th user's hidden states and inputs in the sequence $\mathscr{S}_{i,t}$ as $\mathbf{h}_{user_j}$ and $\mathbf{x}_{user_j}$. The forward formulas are

$$\mathbf{h}_{item_j} = \mathscr{H}_{item}(\mathbf{h}_{item_{j-1}},\mathbf{x}_{item_j}); \quad \mathbf{h}_{user_j} = \mathscr{H}_{user}(\mathbf{h}_{user_{j-1}},\mathbf{x}_{user_j}). \tag{19.4}$$

where $\mathscr{H}_{user}(\cdot,\cdot)$ and $\mathscr{H}_{item}(\cdot,\cdot)$ represent the encoding functions specific to user and item, respectively. We adopt the long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) as the encoder instead of the Transformer (Vaswani et al, 2017),

since LSTM can utilize time feature to control the information to be propagated with the time decay feature as inputs. After the time-aware sequence encoding, we obtain the corresponding hidden states sequence of historical behavior sequence $\mathscr{S}_{u,t}$ of user u and $\mathscr{S}_{i,t}$ of item i. The time-aware sequence encoding functions can be represented as:

$$\begin{aligned} &\text{LSTM}_{item}(\{\mathbf{x}_{i,\tau}|(i,\tau) \in \mathscr{S}_{u,t}\}) = \{\mathbf{h}_{i,\tau}|(i,\tau) \in \mathscr{S}_{u,t}\}; \\ &\text{LSTM}_{user}(\{\mathbf{x}_{u,\tau}|(u,\tau) \in \mathscr{S}_{i,t}\}) = \{\mathbf{h}_{u,\tau}|(u,\tau) \in \mathscr{S}_{i,t}\}. \end{aligned} \tag{19.5}$$

19.2.2.4 Second-Order Graph Attention

In practice, there may exist noisy neighbors, whose interest or audience is irrelevant to the target node. To eliminate the noise brought by the unreliable nodes, we propose an attention mechanism to activate related nodes in the behavior sequence. Traditional graph attention mechanism, like GAT (Veličković et al, 2018), computes attention weights between the central node and the neighbor nodes, which indicate the importance of each neighbor node to the central node. Although they perform well on the node classification task, they may increase noise diffusion for recommendation when there exists an unreliable connection.

To address the above problem, we propose a graph attention mechanism that uses both the parent node of the central node and the central node itself to build the query and takes the neighbor nodes as the key and value. Since we use the parent node of the central node to enhance the expressive power of the query, which is connected to the key node with two hops, we name it *second-order graph attention*. The parent node of the central node can be seen as a complement when the central node is unreliable, thus improving the robustness.

Following the scaled dot-product attention (Vaswani et al, 2017), the attention function is defined as

$$\text{Attention}(Q,K,V) = \frac{\text{softmax}(QK^\top)}{\sqrt{d}}V \tag{19.6}$$

where Q, K and V represent the query, key and value, respectively, and d is the dimension of K and Q. The multi-head attention is defined as follows:

$$\text{MultiHead}(Q,K,V) = [\text{head}_1;\text{head}_2;\ldots;\text{head}_h]W_O \tag{19.7}$$

$$\text{head}_i = \text{Attention}(QW_{Q_i}, KW_{K_i}, VW_{V_i}) \tag{19.8}$$

where weights W_Q, W_K, W_V and W_O are trained parameters.

Given the behavior hidden states sequence $\{\mathbf{h}_{i,\tau}|(i,\tau) \in \mathscr{S}_{u,t}\}$ and $\{\mathbf{h}_{u,\tau}|(u,\tau) \in \mathscr{S}_{i,t}\}$ after the time-aware sequence encoding, we represents the attention process as:

$$\mathbf{x}_{u,t} = \text{ATT}_{item}(\{\mathbf{h}_{i,\tau}|(i,\tau) \in \mathscr{S}_{u,t}\}); \mathbf{x}_{i,t} = \text{ATT}_{user}(\{\mathbf{h}_{u,\tau}|(u,\tau) \in \mathscr{S}_{i,t}\}). \quad (19.9)$$

19.2.2.5 Aggregation and Layer Combination

The core idea of GCN is to learn representation for nodes by performing convolution over their neighborhood. In DSGL, we stack the time-aware sequence encoding and the second-order graph attention, and the aggregator can be represented as:

$$\begin{aligned} \mathbf{x}_{u,t}^{(k+1)} &= \text{ATT}_{item}(\text{LSTM}_{item}(\{\mathbf{x}_{i,t}^{(k)}|i \in \mathscr{S}_{u,t}\})); \\ \mathbf{x}_{i,t}^{(k+1)} &= \text{ATT}_{user}(\text{LSTM}_{user}(\{\mathbf{x}_{u,t}^{(k)}|i \in \mathscr{S}_{i,t}\})). \end{aligned} \quad (19.10)$$

Different from traditional GCN models that use the last layer as the final node representation, inspired by (He et al, 2020), we combine the embeddings obtained at each layer to form the final representation of a user (an item):

$$\hat{\mathbf{x}}_{u,t} = \frac{1}{k_u}\sum_{k=1}^{k_u}\mathbf{x}_{u,t}^{(k)}; \quad \hat{\mathbf{x}}_{i,t} = \frac{1}{k_i}\sum_{k=1}^{k_i}\mathbf{x}_{i,t}^{(k)}, \quad (19.11)$$

where K_u and K_i denote the numbers of DSGL layers for user u and item i, respectively.

19.2.3 Model Prediction

Given an interaction triplet (u,i,t), we can predict the possibility of the user interacting with the item as:

$$\hat{y} = \mathscr{F}(u,i,\mathscr{G}_{u,t}^{(k)},\mathscr{G}_{i,t}^{(k-1)};\Theta) = \text{MLP}([\mathbf{e}_{u,t};\mathbf{e}_{i,t};\hat{\mathbf{x}}_{u,t};\hat{\mathbf{x}}_{i,t}]) \quad (19.12)$$

where $\text{MLP}(\cdot)$ represents the MLP layer and Θ denotes the network parameters. We adopt the cross-entropy loss function:

$$\mathscr{L} = -\sum_{(u,i,t,y)\in\mathscr{D}}[y\log\hat{y} + (1-y)\log(1-\hat{y})] \quad (19.13)$$

where $\mathscr{D}$ is the set of training samples, and $y \in \{0,1\}$ denotes the real label. The algorithm procedure is presented in Algorithm 1.

Algorithm 2 The algorithm of DSGL.

Input:
The training set $\mathscr{D} = \{(u,i,t,y)\}$; User set $\mathscr{U}$; Item set $\mathscr{I}$; Interaction set $\mathscr{E}$; Depths k_u, k_i; Number of epochs E.

Output: Network parameters Θ.

Initialize input feature $\mathbf{f}_{user_u}$ of user $u \in \mathscr{U}$ and $\mathbf{f}_{item_i}$ of item $i \in \mathscr{I}$;
for $e \leftarrow 1$ to E **do**
  for $(u,i,t,y) \in \mathscr{D}$ **do**
    Construct DSGs $\mathscr{G}_{u,t}^{(k_u)}$, $\mathscr{G}_{i,t}^{(k_i)}$ for user u and item i from $\mathscr{E}$;
    for $(v,j,\tau) \in \mathscr{G}_{u,t}^{(k_u)} \bigcup \mathscr{G}_{i,t}^{(k_i)}$ **do**
      Obtain the behavior sequence $\mathscr{S}_{v,\tau}$ and $\mathscr{S}_{j,\tau}$;
      $\mathbf{x}_{v,\tau}^{(0)} \leftarrow \mathbf{e}_{v,\tau}$; $\mathbf{x}_{j,\tau}^{(0)} \leftarrow \mathbf{e}_{j,\tau}$;
      for $k \leftarrow 1$ to k_u **do**
        $\mathbf{x}_{v,\tau}^{(k)} \leftarrow \mathrm{ATT}_{item}(\mathrm{LSTM}_{item}(\{\mathbf{x}_{j,\tau}^{(k-1)} | i \in \mathscr{S}_{v,\tau}\}))$;
      end for
      for $k \leftarrow 1$ to k_i **do**
        $\mathbf{x}_{j,\tau}^{(k)} \leftarrow \mathrm{ATT}_{user}(\mathrm{LSTM}_{user}(\{\mathbf{x}_{v,\tau}^{(k-1)} | i \in \mathscr{S}_{j,\tau}\}))$;
      end for
    end for
    $\hat{\mathbf{x}}_{u,t} \leftarrow \frac{1}{k_u}\sum_{k=1}^{k_u} \mathbf{x}_{u,t}^{(k)}$; $\hat{\mathbf{x}}_{i,t} \leftarrow \frac{1}{k_i}\sum_{k=1}^{k_i} \mathbf{x}_{i,t}^{(k)}$;
    $\hat{y}_{u,i,t} \leftarrow \mathrm{MLP}([\mathbf{e}_{u,t}; \mathbf{e}_{i,t}; \hat{\mathbf{x}}_{u,t}; \hat{\mathbf{x}}_{i,t}])$;
    Update the parameters Θ by optimizing Eq.19.13;
  end for
end for=0

19.2.4 Experiments and Discussions

We evaluate our methods on the real-world Amazon product datasets[2], and use five subsets. The widely used metrics for the CTR prediction task, i.e., AUC (the area under the ROC curve) and Logloss, are adopted. The compared recommendation methods can be grouped into five categories, including conventional methods (SVD++ (Koren, 2008) and PNN (Qu et al, 2016)), sequential methods with user behaviors (GRU4Rec (Hidasi et al, 2015), CASER (Tang and Wang, 2018), ATRANK (Zhou et al, 2018a) and DIN (Zhou et al, 2018b)), sequential methods with user and item behaviors (Topo-LSTM (Wang et al, 2017b), TIEN (Li et al, 2020g) and DIB (Guo et al, 2019a)), static-graph-based methods (NGCF (Wang et al, 2019k) and LightGCN (He et al, 2020)), and dynamic-graph-based method (SR-GNN (Wu et al, 2019c)).

19.2.4.1 Performance Comparison

To demonstrate the performance of the proposed model, we compare DSGL with the state-of-the-art recommendation methods. We find that DSGL consistently out-

[2] http://snap.stanford.edu/data/amazon/productGraph/

performs all other baselines, demonstrating its effectiveness. The sequential models outperform the conventional methods by a large margin, proving the effectiveness of capturing temporal dependency in recommendation. The sequential methods which model both user behaviors and item behaviors outperform the methods that only use the user behavior sequences, which verifies the importance of both user- and item-side behavior information. The performance of the static-graph-based methods, including LightGCN and NGCF, are not competitive. The reasons are two folds. First, these methods ignore the new interactions in the testing set in the inference phase. Second, since they do not model the temporal dependency of interactions, they cannot capture the evolving interests, degrading the performances compared with sequential models. The session-graph-based method SR-GNN outperforms static-graph-based methods, because SR-GNN incorporates all the interacted items before the current moment into graphs dynamically. However, it underperforms the sequential methods. One possible reason could be that the ratio of repeated items in the sequences is low in the Amazon datasets, and the transitions of items are not complex enough to be modeled as graphs.

19.2.4.2 Effectiveness of Graph Structure and Layer Combination

To show the effectiveness of the graph structure and layer combination, we compare the performance of **DSGL** and its variant **DSGL w/o LC** that uses the last layer instead of the combined layer as the final representation w.r.t different numbers of layers. Focusing on DSGL with layer combination, the performance gradually improves with the increase of layers. We attribute the improvement to the collaborative information carried by the second-order and third-order connectivity in the graph structure. Comparing DSGL and DSGL w/o LC, we find that removing the layer combination degrades the performance largely, which demonstrates the effectiveness of layer combination.

19.2.4.3 Effectiveness of Time-Aware Sequence Encoding

In DSGL, we perform time-aware sequence encoding to preserve both the order of behaviors and the time information. Thus, we design ablation experiments to study how the temporal dependency and time information in DSGL contributes to the final performance. To evaluate the role of time information, we test the removal of time feature only of the item bahavior (i.e., **DSGL w/o time in UBH**), of the user behavior (i.e., **DSGL w/o time in IBH**), and of both behaviors (i.e., **DSGL w/o time**). To evaluate the contribution of the behavior order, we test the removal of the sequence encoding module while retaining time information (i.e., **DSGL w/o Seq ENC**) and the removal of the time-aware sequence encoding (i.e., **DSGL w/o TA Seq ENC**). From the comparison, we find that DSGL outperforms DSGL w/o TA Seq ENC by a significant margin, demonstrating the efficacy of the time-aware sequence encoding layer. Comparing DSGL w/o time, DSGL w/o time in UBH and

DSGL w/o time in IBH with the default DSGL, we observe that removing the time information on either user or item behavior side will cause performance degradation. DSGL outperforms DSGL w/o Seq ENC, confirming the importance of temporal dependency carried by the historical behavior sequence.

19.2.4.4 Effectiveness of Second-Order Graph Attention

In DSGL, we propose a second-order graph attention to eliminate noise from unreliable neighbors. To justify its rationality, we explore different choices here. We test the performance without graph attention (i.e., **DSGL w/o ATT**). We also replace the second-order graph attention with the traditional graph attention (i.e., **DSGL-GAT**). Note that the attention function in DSGL-GAT here is the same as the one in DSGL, and the only difference is the query. DSGL-GAT takes the central node as the query. From the results, we have the following observations:

- The best setting in all cases is adopting the second-order graph attention (i.e., the current design of DSGL). Replacing it with GAT drops the performance, demonstrating the effectiveness of second-order attention in activating related neighbors and eliminating the noise from reliable neighbors.
- Removing the attention mechanism (i.e., DSGL w/o ATT), the performance degrades largely, worse than DSGL with traditional graph attention. In some cases, the performance is even not as good as the best baseline. The observation demonstrates the necessity to introduce the attention mechanism in GNN-based recommendation methods due to the inevitable noise in the multi-hop neighborhood.

19.3 Case Study 2: Device-Cloud Collaborative Learning for Graph Neural Networks

19.3.1 The proposed framework

Recently, several works (Sun et al, 2020e; Cai et al, 2020a; Gong et al, 2020; Yang et al, 2019e; Lin et al, 2020e; Niu et al, 2020) have explored the on-device computing advantages in recommender systems. This drives the development of on-device GNNs, e.g., DSGL in the previous section. However, these early works either only consider the cloud modeling, or on-device inference, or the aggregation of the temporal on-device training pieces to handle the privacy constraint. Little has explored the device modeling and the cloud modeling jointly to benefit both sides for GNNs. To bridge this gap, we introduce a Device-Cloud Collaborative Learning framework as shown in Figure 23.2. Given a recommendation dataset $\{(\mathbf{x}_n, y_n)\}_{n=1,\ldots,N}$, we target to learn a GNN-based mapping function $f : \mathbf{x}_n \rightarrow y_n$ on the cloud side. Here, $\mathbf{x}_n$ is the graph feature that contains all available candidate features and user context, y_n is the user implicit feedback (click or not) to the corresponding candidate and N is the

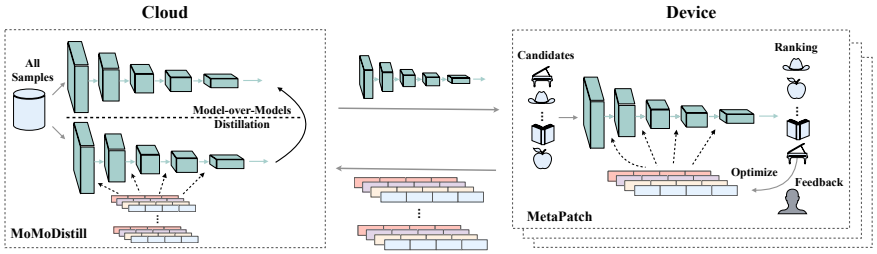


Fig. 19.3: The general DCCL framework for recommendation. The cloud side is responsible to learn the centralized cloud GNN model via the model-over-models distillation from the personalized on-device GNN models. The device receives the cloud GNN model to conduct the on-device personalization. We propose *MoMoDistill* and *MetaPatch* to instantiate each side respectively.

sample number. On the device side, each device (indexed by m) has its own local dataset, $\left\{(\mathbf{x}_n^{(m)}, y_n^{(m)})\right\}_{n=1,\ldots,N^{(m)}}$. We add a few parameter-efficient patches (Yuan et al, 2020a) to the cloud GNN model f (freezing its parameters on the device side) for each device to build a new GNN $f^{(m)} : \mathbf{x}_n^{(m)} \rightarrow y_n^{(m)}$. In the following, we will present the practical challenges in the deployment and our solutions.

19.3.1.1 MetaPatch for On-device Personalization

Although the device hardware has been greatly improved in the recent years, it is still resource-constrained to learn a complete big model on the device. Meanwhile, only finetuning last few layers is performance-limited due to the feature basis of the pretrained layers. Fortunately, some previous works have demonstrated that it is possible to achieve the comparable performance as the whole network finetuning via patch learning (Cai et al, 2020b; Yuan et al, 2020a; Houlsby et al, 2019). Inspired by these works, we insert the model patches on basis of the cloud model f for on-device personalization. Formally, the output of the l-th layer attached with one patch on the m-th device is expressed as

$$f_l^{(m)}(\cdot) = f_l(\cdot) + \mathbf{h}_l^{(m)}(\cdot) \circ f_l(\cdot), \tag{19.14}$$

where LHS of Eq.19.14 is the sum of the original $f_l(\cdot)$ and the patch response of $f_l(\cdot)$. Here, $\mathbf{h}_l^{(m)}(\cdot)$ is the trainable patch function and $\circ$ denotes the function composition that treats the output of the previous function as the input. Note that, the model patch could have different neural architectures. Here, we do not explore its variants but specify the same bottleneck architecture like (Houlsby et al, 2019).

Nevertheless, we empirically find that the parameter space of multiple patches is still relatively too large and easily overfits the sparse local samples. To overcome

this issue, we propose *MetaPatch* to reduce the parameter space. It is a kind of meta learning methods to generate parameters (Ha et al, 2017; Jia et al, 2016). Concretely, assume the parameters of each patch are denoted by $\theta_l^{(m)}$ (flatten all parameters in the patch into a vector). Then, we can deduce the following decomposition

$$\theta_l^{(m)} = \Theta_l * \hat{\theta}^{(m)}, \tag{19.15}$$

where Θ_l is the globally shared parameter basis (freezing it on the device and learned in the cloud) and $\hat{\theta}^{(m)}$ is the surrogate tunable parameter vector to generate each patch parameter $\theta_l^{(m)}$ in the device-GNN-model $f^{(m)}$. To facilitate the understanding, we term $\hat{\theta}^{(m)}$ as the metapatch parameter. In this paper, we keep the number of patch parameters is greatly less than that of the metapatch parameters to be learned for personalization. Note that, regarding the pretraining of Θ_l, we leave the discussion in the following section to avoid the clutter, since it is learned on the cloud side. According to Eq. 19.15, we implement the patch parameter generation via the metapatch parameter $\hat{\theta}^{(m)}$ instead of directly learning $\theta^{(m)}$. To learn the metapatch parameter, we can leverage the local dataset to minimize the following loss function.

$$\min_{\hat{\theta}^{(m)}} \ell(y, \tilde{y})\big|_{\tilde{y}=f^{(m)}(\mathbf{x})}, \tag{19.16}$$

where ℓ is the pointwise cross-entropy loss, $f^{(m)}(\cdot) = f_L^{(m)}(\cdot) \circ \cdots f_l^{(m)}(\cdot) \cdots \circ f_1^{(m)}(\cdot)$ and L is the number of total layers. After training the device specific parameter $\hat{\theta}^{(m)}$ by Eq. 19.16, we can use Eq. 19.15 to generate all patches, and then insert them into the cloud GNN model f via Eq. 19.14 to get the final personalized GNN model $f^{(m)}$, which will provide the on-device personalized recommendation.

19.3.1.2 MoMoDistill to Enhance the Cloud Modeling

The conventional incremental training of the centralized cloud model follows the "model-over-data" paradigm. That is, when the new training samples are collected from devices, we directly perform the incremental learning based on the model trained in the early sample collection. The objective is formulated as follows,

$$\min_{W_f} \ell(y, \hat{y})\big|_{\hat{y}=f(\mathbf{x})}, \tag{19.17}$$

where W_f is the network parameter of the cloud GNN model f to be trained. This is an independent perspective without considering the device modeling. However, the on-device personalization actually can be more powerful than the centralized cloud model to handle the corresponding local samples. Thus, the guidance from the on-device models could be a meaningful prior to help the cloud modeling. Inspired by this, we propose a "model-over-models" paradigm to simultaneously learn from data and aggregate the knowledge from on-device models, to enhance the training of the centralized cloud model. Formally, the objective with the distillation procedure

on the samples from all devices is defined as,

$$\min_{W_f} \ell(y,\hat{y}) + \beta \ \mathrm{KL}(\tilde{y},\hat{y})\big|_{\hat{y}=f(\mathbf{x}),\tilde{y}=f^{(m)}(\mathbf{x})}, \tag{19.18}$$

where β is the hyperparameter to balance the distillation and "model-over-data" learning. Note that, the feasibility of the distillation in Eq. 19.18 critically depends on the patch mechanism in the previous section, since it allows us to input the meta-patch parameters like features with only loading the other parameters of $f^{(m)}$ in one time. Otherwise, we will suffer from the engineering issue of reloading numerous checkpoints frequently, which is almost impossible for current frameworks.

In *MetaPatch*, we introduce the global parameter basis $\{\Theta_l\}$ (simplified by Θ) to reduce the parameter space on the device. Regarding its training, we empirically find that coupled learning with W_f easily falls into undesirable local optimal, since they play different roles in terms of their semantics. Therefore, we resort to a progressive optimization strategy, that is, first optimize f based on Eq. 19.18, and then distill the knowledge for the parameter basis Θ with the learned f. For the second step, we design an auxiliary component by considering the heterogeneous characteristics of the metapatches from all devices and the cold-start issue at the beginning. Concretely, given the dataset $\{(x,y,\mathbf{u}^{(I(x))},\hat{\theta}^{(I(x))})\}_{n=1,\ldots,N}$, where I maps the sample index to the device index and $\mathbf{u} \subset x$ is the user profile features (*e.g.*, age, gender, purchase level, etc) of the corresponding device, we define the following auxiliary encoder,

$$U(\hat{\theta},u) = W^{(1)}\tanh(W^{(2)}\hat{\theta} + W^{(3)}\mathbf{u}), \tag{19.19}$$

where $W^{(1)}$, $W^{(2)}$, $W^{(3)}$ are tunable projection matrices. Here, we use W_e denoting the collection $\{W^{(1)},W^{(2)},W^{(3)}\}$ for simplicity. To learn the global parameter basis, we replace $\hat{\theta}$ by $U(\hat{\theta},\mathbf{u})$ to simulate Eq. 19.15 to generate the model patch, *i.e.*, $\Theta * U(\hat{\theta},\mathbf{u})$, since actually $\hat{\theta}$ is too heterogeneous to be directly used. Then, combining $\Theta * U(\hat{\theta},\mathbf{u})$ with f learned in the first distillation step, we can form a new proxy device model $\hat{f}^{(m)}$ (different from $f^{(m)}$ in the patch generation). Here, we leverage such a proxy $\hat{f}^{(m)}$ to directly distill the knowledge from the true $f^{(m)}$ collected from devices, which optimizes Θ and the parameters of the auxiliary encoder,

$$\min_{(\Theta,\, W_e)} \ell(y,\hat{y}) + \beta \ \mathrm{KL}(\tilde{y},\hat{y})\big|_{\hat{y}=\hat{f}^{(m)}(x),\tilde{y}=f^{(m)}(x)}, \tag{19.20}$$

Eq. 19.18 and Eq. 19.20 progressively help learn the centralized cloud model and the global parameter basis. We specially term this progressive distillation mechanism as *MoMoDistill* to emphasize our "model-over-models" paradigm different from the conventional "model-over-data" incremental training on the cloud side. Finally, in Algorithm 3, we summarize the complete procedure of DCCL.

Algorithm 3 Device-Cloud Collaborative Learning for GNNs

Pretrain the cloud GNN model f, and then learn the global parameter basis Θ based on Eq. 19.20 by setting $\hat{\theta}$ as 0.

while lifecycle **do** Send f and Θ to devices.
Device(f, Θ): ▷ *MetaPatch*
1) Accumulate the local data into batches
2) On-device personalization via Eq.19.16
3) If time $>$ threshold: upload personalized GNN model $f^{(m)}$
4) Else: return the step 1).
Recycle all model patches $\{\hat{\theta}^{(m)}\}$.
Cloud($\{\hat{\theta}^{(m)}\}$): ▷ *MoMoDistill*
1) Optimize the cloud GNN model f based on Eq.19.18
2) Learn the parameter basis Θ by Eq.19.20

19.3.2 Experiments and Discussions

To demonstrate the effectiveness of the proposed framework, we conduct a range of experiments on three recommendation datasets Amazon, Movielens-1M and Taobao. Generally, all these three datasets are user interactive history in sequence format, and the last user interacted item is cut out as test sample. For each last interacted item, we randomly sample 100 items that do not appear in the user history. We compare our framework with some classical cloud models, namely, the conventional methods MF (Koren et al, 2009) and FM (Rendle, 2010), deep learning-based methods NeuMF (He et al, 2017b) and DeepFM (Guo et al, 2017), and sequence-based methods SASRec (Kang and McAuley, 2018) and DIN (Zhou et al, 2018b). For the whole experiments, we implement our model on the basis of DIN, where we insert the model patches in the last second fully-connected layer and the first two fully-connected layers after the feature embedding layer. In all comparisons, we term *MetaPatch* as DCCL-e, and *MoMoDistill* as DCCL-m, since the whole framework resembles EM iterations. The default method to compare the baselines is named DCCL, which indicates that it goes through both on-device personalization and the "model-over-models" distillation. The performance are measured by HitRate, NDCG and macro-AUC.

19.3.2.1 How is the performance of DCCL compared with the SOTAs?

To demonstrate the effectiveness of DCCL, we conduct the experiments on Amazon, Movielens and Taobao to compare to a range of baselines. Aligned with the popular experimental settings (He et al, 2017b; Zhou et al, 2018b), the last interactive item of each user on three datasets is left for evaluation and all items before the last one are used for training. For DCCL, we split the training data into two parts on average according to the temporal order: one part is for the pretraining

of the backbone (DIN) and the other part is for the training of DCCL. In the experiments, we conduct one-round DCCL-e and DCCL-m. Finally, the DCCL-m is used to compare with the six representative models. We find that the deep learning based methods NeuMF and DeepFM usually outperform the conventional methods MF and FM, and the sequence-based methods SASRec and DIN consistently outperform previous non-sequence-based methods. Our DCCL builds upon on the best baseline DIN and further improves its results. Specifically, DCCL shows about 2% or more improvements in terms of NDCG@10, and at least 1% improvements in terms of HitRate@10 on all three datasets. The performances on both small and large datasets confirm the superiority of our DCCL.

19.3.2.2 Whether on-device personalization benefits to the cloud model?

In this section, we target to demonstrate that how on-device personalization via *MetaPatch* (abbreviated as DCCl-e) can improve the recommendation performance from different levels of users compared with the centralized cloud model. Considering the data scale and the availability of the context information for visualization, only the Taobao dataset is used to conduct this experiment. To validate the performance of DCCL-e in the fine-grained granularity, we sort the users based on their sample numbers and then partition them into 20 groups on average along the sorted user axis (see the statistic of the sample number *w.r.t.* the user in the appendix). After on-device model personalization, we calculate the performance for each group based on the personalized models. Here, the macro-AUC metric is used, which equally treats the users in the group instead of the group AUC in (Zhou et al, 2018b).

We use DIN as baseline and pretrain it on the Taobao Dataset of the first 20 days. Then, we test the model in the data of the remaining 10 days. For DCCL-e, we first pretrain DIN on the Taobao Dataset of the first 10 days, and then insert the patches into the pretrained DIN same as previous settings. Finally, we perform the on-device personalization in the subsequent 10 days. Similarly, we test DCCL-e on the data of the last 10 days. The evaluation is respectively conducted in the 20 groups. According to the results, we find that with the increase of the group index number, the performance approximately decreases. This is because the users in the group of larger indices are more like the long-tailed users based on our partition, and their patterns are easily ignored or even sacrificed by the centralized cloud model. In comparison, DCCL-e shows the consistent improvement over DIN on all groups, and especially can achieve a large improvement in long-tailed user groups.

19.3.2.3 The iterative characteristics of the multi-round DCCL.

To illustrate the convergence property of DCCL, we conduct the experiments on the Taobao dataset in different device-cloud interaction temporal intervals. Concretely, we specify every 2, 5, 10 days interactions between device and cloud, and respec-

tively trace the performance of each round evaluated on the last click of each user. According to the results, we observe that frequent interactions achieve much better performance than the infrequent counterparts. We speculate that, as *MeatPatch* and *MoMoDistill* could promote each other at every round, the advantages in performance have been continuously strengthened with more frequent interactions. However, the side effect is we have to frequently update the on-device models, which may introduce other uncertain crash risks. Thus, in the real-world scenarios, we need to make a trade-off between performance and the interaction interval.

19.3.2.4 Ablation Study of DCCL

For the first study, we given the results of the one-round DCCL on the Taobao dataset and compare with DIN. From the results, we can observe the progressive improvement after DCCL-e and DCCL-m, and DCCL-m acquires more benefit than DCCL-e in terms of the improvement. The revenue behind DCCL-e is *MetaPatch* customizes a personalized model for each user to improve their recommendation experience once new behavior logs are collected on device, without the delayed update from the centralized cloud server. The further improvements from DCCL-m confirm the necessity of *MoMoDistill* to re-calibrate the backbone and the parameter basis in a long term. However, if we conduct the experiments without our two modules, the model performance is as DIN, which is not better than DCCL.

For the second ablation study, we explore the effect of the model patches in different layer junctions. In previous sections, we insert two patches (1st Junction, 2nd Junction) in the two fully-connected layers respectively after the feature embedding layer, and one patch (3rd Junction) to the layer before the last softmax transformation layer. In this experiment, we validate their effectiveness by only keep each of them in one-round DCCL. Compared with the full model, we can find that removing the model patch would decrease the performance. The results suggest the patches in the 1st and 2nd junctions are more effective than the one in the 3rd junction.

19.4 Future Directions

Certainly, we have witnessed the arising trends for GNNs to be applied in various areas. We believe the following directions should be paid more attention for GNNs to have wider impacts in big data areas, especially in search, recommendation or advertisement.

- There is still a lot to understand about GNNs, but there were quite a few important results about how they work (Loukas, 2020; **?**; Oono and Suzuki, 2020). Future research works of GNNs should balance between technical simplicity, high practical impact, and far-reaching theoretical insights.
- It is also great to see how GNNs can be applied for other real-world tasks (Wei et al, 2019; Wang et al, 2019a; Paliwal et al, 2020; Shi et al, 2019a; Jiang and

Balaprakash, 2020; Chen et al, 2020o). For example, we see applications in fixing bugs in Javascript, game playing, answering IQ-like tests, optimization of TensorFlow computational graphs, molecule generation, and question generation in dialogue systems, among many others.

- It will become popular to see GNNs applied for knowledge graph reasoning (Ren et al, 2020; Ye et al, 2019b). A knowledge graph is a structured way to represent facts where nodes and edges actually bear some semantic meaning, such as the name of the actor or act of playing in movies.
- Recently there are new perspectives on how we should approach learning graph representations, especially considering the balance between local and global information. For example, Deng et al (2020) presents a way to improve running time and accuracy in node classification problem for any unsupervised embedding method. Chen et al (2019c) shows that if one replaces a non-linear neighborhood aggregation function with its linear counterpart, which includes degrees of the neighbors and the propagated graph attributes, then the performance of the model does not decrease. This is aligned with previous statements that many graph data sets are trivial for classification and raises a question of the proper validation framework for this task.
- Algorithmic works of GNNs should be integrated with system design more closely, to empower end-to-end solutions for users to address their scenarios by taking graph into deep learning frameworks. It should allow pluggable operators to adapt to the fast development of GNN community and excels in graph building and sampling. As an independent and portable system, the interfaces of AliGraph (Zhu et al, 2019c) can be integrated with any tensor engine that is used for expressing neural network models. By co-designing the flexible Gremlin like interfaces for both graph query and sampling, users can customize data accessing pattern freely. Moreover, AliGraph also shows excellent performance and scalability.

Editor's Notes: Recommender system is one of the hottest topics in both research and industrial communities due to its huge value in a number of commercial businesses such as Amazon, Facebook, LinkedIn, and so on. Since user-item interactions, user-user interaction and item-item similarity can naturally formulate into graph structure data, various graph representation learning techniques (GNN Methods in Chapter 4, GNN Scalability in Chapter 6, Graph Structure Learning in Chapter 14, Dynamic GNNs in Chapter 15, and Heterogeneous GNNs in Chapter 16) can serve a strong set of algorithmic foundations in applying GNNs for developing an effective and efficient modern recommendation system.