ABSTRACT

In the past few years, recommender systems have become increasingly popular in marketing and have caught the attention of commercial companies because of their ability to target potential customers and even make predictions based on their historical behavior. Traditional recommender systems usually predict user ratings for specific items, so matrix factorization techniques can effectively express user and item features.

However, with the popularity of multi-modal and high-dimensional data, matrix factorization techniques are far from sufficient. In practical applications, we often have to deal with multiple relationships. For example, in online marketing, we will concentrate on the multiple relationships between users, items, and time. Based on that, we are able to understand the temporal dynamics of user preferences and utilize them to provide recommendations to users.

Fortunately, tensors (multi-dimensional extensions of matrices) are very powerful data structures that can represent and model such relationships. To further capture the underlying factors of tensors, there already exist various tensor decomposition algorithms developed for high-dimensional data. Furthermore, most widely applied tensor decomposition methods assume that tensors with missing values have low rank. In other words, a low-rank tensor decomposition divides a tensor with missing values into several smaller parts (the core tensor and corresponding factor matrix in the Tucker decomposition). We are then able to recover the tensor and predict missing values, even if we only have limited observations of the tensor. However, these tensor decomposition techniques will suffer from the data sparsity issue. When tensors are sparse, prediction accuracy tends to be much lower. To alleviate this problem, I decide to incorporate auxiliary information of users and items into the rating dataset.

In my opinion, the problem lies in how to effectively incorporate user and item's auxiliary information. To solve this problem, this work proposes a modified tensor decomposition method based on the tucker decomposition. On the one hand, I regard the auxiliary information as a constraint on the factor matrix in the tucker decomposition to ensure that the representations of users and items learned through the tucker decomposition are close enough to the *side information*. On the other hand, *side information* can function as a regularization technique that helps alleviate over-fitting problems caused by the data sparsity issue. Therefore, this tensor completion method is then formulated as a constrained optimization problem, which can be transformed into an unconstrained optimization problem with the help of the Lagrange multiplier technique. The gradient of the core tensor and the factor matrix of each dimension can be obtained by calculating the partial derivative of the objective function. Gradient descent is then applied to find the optimal solution, and the performance of the proposed tensor completion method on the MovieLens dataset has been evaluated.

Having obtained representations of users, items and time through the modified tucker decomposition, I further explored methods for predicting user ratings over specific items. Inspired by [1], I view tensor completion for recommender systems as link prediction on multi-dimensional graphs. Specifically, interactions such as movie ratings can be represented by a series of bipartite graphs with labeled edges denoting the corresponding ratings. Moreover, representing movie ratings as a bipartite graph works well for my task when given auxiliary information of users and items. In recent years, increasing attention has been paid on generalizing convolutional neural networks (CNNs) to graphs with the purpose of harnessing the power of representation-learning, and several approaches have been demonstrated to improve performances on node-level tasks such as link prediction. The biggest challenge is that most graph convolutional networks (GCNs) are designed for single-dimensional graphs, with only one type of relationship between a pair of nodes. Obviously, they cannot consider multiple types of relationships simultaneously. If we try to perform message passing at each layer independently and compress them into one graph in the end, the structure of the tensors will be broken, resulting in low prediction accuracy. In order to preserve the properties of different graphs, I adopt an approach which is able to process complex graphs with multiple relationships. Each type of relationships forms a graph, and these graphs are not independent of each other. Instead, the relationships of different dimensions are closely linked. In [2] and [3], they both propose a framework to capture intra- and inter-dimension interactions, which can reconcile heterogeneous relationships from multiple graphs and model rich information in multi-dimensional graphs coherently for representation learning. Specifically, they both introduce a concept called **virtual graph** for merging heterogeneous information across dimensions and performing inter-graph message passing. In that framework, virtual graphs are regarded as undirected graphs because different graphs and dimensions have the same correlation with link prediction. However, their method seems unsuitable for predicting future user ratings, so I modified their inter-graph messaging strategy and the way virtual graphs are constructed based on real-world problems.

KEY WORDS: Tensor Decomposition; Side Information; Multi-dimensional Graph Convolutional Networks; Recommendation System

CONTENTS

1 Introduction

In the past few years, recommender systems have become increasingly popular in online marketing and drawn commerce companies' attention, since they are able to target potential customers and even make predictions based on customers' historical behavior. Traditional recommender systems typically predict the rating of a user for a particular item, and thus matrix decomposition techniques could effectively express users and items' embeddings.

However, with the prevalence of multi-modal data and high-dimensional data, matrix decomposition techniques would be insufficient. In real applications, we often have to deal with multiple relationships. For instance, in online marketing, multi-relationships among users, items and time are the focus of our interest, based on which we are able to learn temporal dynamics of users' preferences and utilize them to make recommendations for users.

Fortunately, tensor (the multi-dimensional extension of matrices), is a quite powerful data structure to represent and model such multi-relationships. To further capture the latent factors underlying tensor, various tensor decomposition algorithms have been developed for high-dimensional data. Also, most of the widely used tensor decomposition methods would assume the tensor with missing values has low rank. In other words, low-rank tensor decomposition divides a tensor with missing values into several smaller parts (e.g. a core tensor and corresponding factor matrices for each dimension in tucker decomposition). Then we are able to recover the tensor and predict the missing values even if we are given only a subset of entries. Nevertheless, these tensor decomposition techniques will suffer from the data sparsity issue. When observations are sparse, predictive accuracy tends to be much lower. For more understandable, Figure 1-1 shows the prediction errors by CP-decomposition against the fraction of unobserved entries for a particular tensor. In order to alleviate this issue, I decide to incorporate auxiliary information on the users and items themselves (also known as *side information*) into the rating data.

From my perspective, the problem lies in how to incorporate the *side information* of both users and items. To solve this problem, I propose a modified tensor decomposition method based on tucker decomposition in this paper. On the one hand, I regard the *side information* as constraints on the factor matrices in tucker decomposition to guarantee that the users and items' representations learned through tucker decomposition are close enough to auxiliary information. On the other hand, the *side information* could serve as the regularization technique and help to alleviate the overfitting problem caused by the data sparsity issue. Therefore, the tensor completion method using *side information* based on tucker decomposition is formulated as a constrained optimization problem, which could be solved with the help of the Lagrange

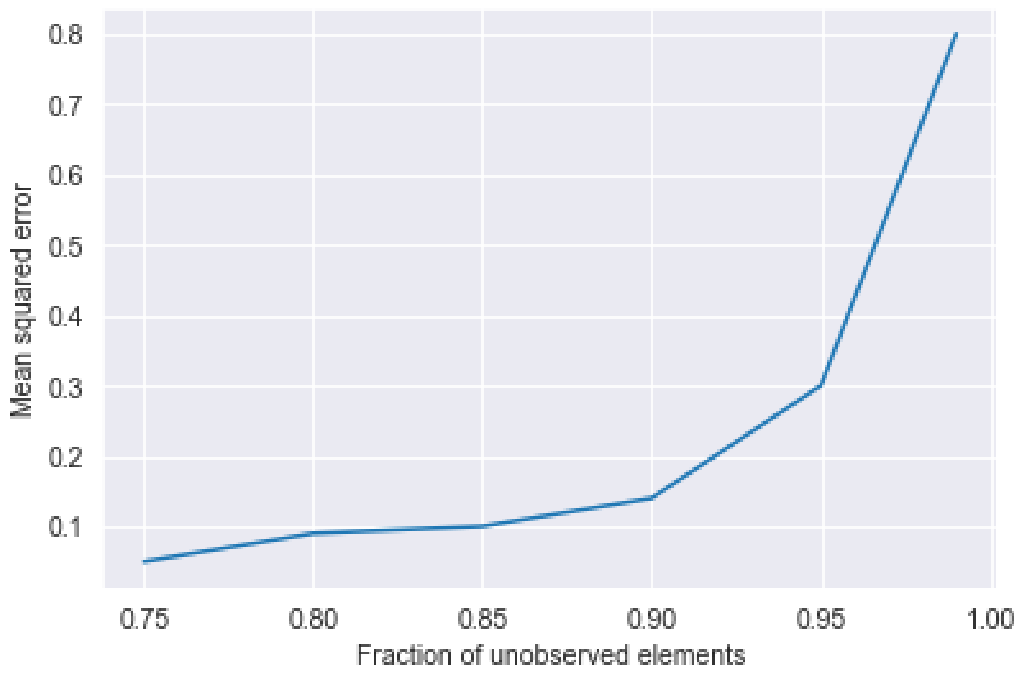

Figure 1-1 The tensor completion performance by CP-decomposition

multiplier technique. The gradient of core tensor $\mathcal{G}$ as well as factor matrix $A_i,\ i = 1, 2, ..., n$ for each dimension can be obtained by computing the partial derivatives of objective function. And gradient descent is applied to find the optimal solution and I evaluate the performance of the proposed tensor completion method on MovieLens dataset [4] by experiments.

Having obtained user, item and time's representation through modified tucker decomposition, I further explore methods to predict users' ratings over a particular item. Motivated by [1], I consider tensor completion for recommender systems as link prediction on multi-dimensional graphs. Specifically, interaction data like movie ratings could be represented by a series of bipartite user-item graph with labeled edges denoting the corresponding ratings. What's better, representing movie ratings as bipartite graphs would fit my task like a glove, given the accessibility of additional users and items' *side information*. In recent years, increasing attention has been paid on generalizing convolutional neural networks (CNNs) to graph to leverage the great power in learning representation, and some methods have been shown to advance the performance of node-level tasks such as link prediction. The greatest challenge is that the majority of graph convolutional networks (GCNs) are designed for single dimensional graphs with only one type of relation between a pair of nodes. Apparently, they cannot consider multiple types of relations simultaneously. If we try to perform message passing on each layer independently and squeeze them into one graph, the structure of the tensor would be destroyed leading to low prediction accuracy. In order to maintain different graphs' properties, I employed a method [2]

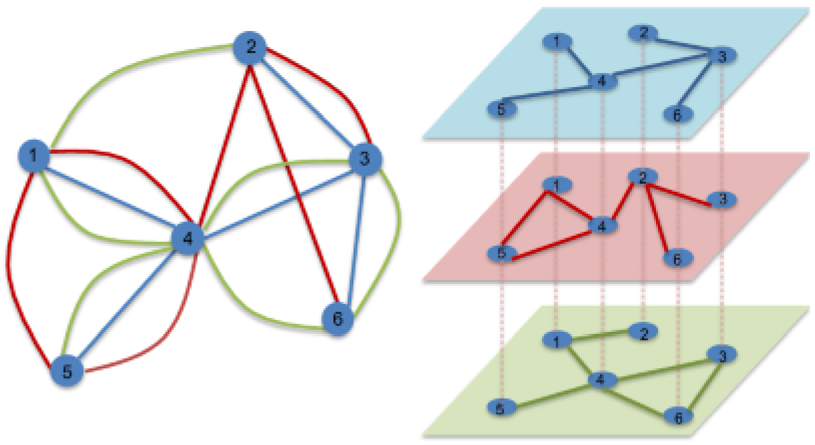

Figure 1-2 Single-dimensional graph and multi-dimensional graph

which treat complex graphs with multiple relations as multi-dimensional graphs as shown in Figure 1-2. Each kind of relationship would form a graph structure, and these graph layers are not independent of each other. Instead, relationships in different dimensions are closely connected. In [2] and [3], both of them propose a framework to capture the interactions within as well as across dimensions, which could harmonize heterogeneous relationships from multiple graphs, and model rich information in multi-dimensional graphs coherently for representation learning. To be more specific, they both introduce a concept called *virtual graph*, which is used for incorporate heterogeneous information across dimensions and perform inter-graph message passing. In their framework, the *virtual graphs* are deemed as undirected graphs, since different graphs and dimensions bear the same relevance to link prediction. However, their method seem not suitable for making predictions on users' rating in the future, so I modify their inter-graph message passing strategy and the construction of the *virtual graph* based on the practical issues.

In this work, I study the problem of making predictions based on users' historical rating behavior, utilizing tensor decomposition with *side information* as well as graph convolutional networks for multi-dimensional graphs. Essentially, I aim to tackle the following problems:

- how to effectively learn representations of each point in time given users and items' structured *side information*;
- how to model the interactions both within and across different dimensions;
- how to incorporate heterogeneous information using multi-dimensional graph convolutional networks and predict users' rating over a particular item with user and item's embedding.

These three problems are addressed by the modified Tucker decomposition, attention-based time weighted inter-graph message passing strategy as well as the proposed framework *Temporal GCN*. My main contributions could be summarized as follows:

- I propose a modified tucker decomposition regarding users and items' *side information* as constraints or regularization terms and solve this optimization problem through gradient descent;
- I propose a multi-dimensional graph convolutional network framework *Temporal GCN*, which could consider heterogeneous information across dimensions simultaneously;
- I introduce a directed *virtual graph* to take users' past rating information into account when make predictions on the users' future ratings;
- I combine representations of each point in time learned from tucker decomposition and a kind of attention function $p_{g,d} = att(w_g, w_d) = w_g^T M w_d$, where M is the parameters to be learned in the bilinear function, aiming to design an appropriate inter-graph message passing strategy;
- I conduct comprehensive experiments on MovieLens dataset, constructing rating tensor, decomposing into smaller parts and performing intra- and inter-graph message passing to demonstrate the effectiveness of the proposed framework.

The rest of my work is organized as follows:

- In section 2, I introduce the notations and definitions I will user in the remaining of this work;
- In section 3, I talk about some relevant fundamental theories concerning the n-mode product (the multiplication between a tensor and a matrix), tucker decomposition and graph convolutional networks;
- In section 4, I give details about the approach to incorporate heterogeneous information across dimensions and model the interactions within and between different graphs;
- In section 5, I describe the experiment setting, and analyze the experimental results;
- I will conclude my work in section 6 together with possible future directions.

2 Definitions and notations

Before I talk about my proposed framework, I'd like to first introduce some notations and definitions I will use in the rest of my work. Below Table 2-1 provides a list of notations and definitions concerning tensor, and Table 2-2 provides a list of notations concerning the graph representation of rating matrix (or rating tensor) as well as graph convolutional networks.

2.1 Tensor operation

Two-dimensional matrix decomposition techniques cannot represent recommendation task involving multiple relationships between users and items, such as user-item rating data over time. As a result, classic matrix decomposition algorithms should be extended to three-dimensional tensor decomposition.

Utilizing various tensor decomposition techniques, we are allowed to decompose user-item-temporal rating data into smaller parts and map them into a joint latent vector space. Therefore, we could obtain users' preferences, items and time's properties as well as the weight of their ternary interactions *(user, item, time)*. And tucker decomposition is widely used as a tensor decomposition paradigm, which is a high-dimensional extension of singular value decomposition of two-dimensional matrix. Here I denote the three-dimensional rating tensor as $\mathcal{X}$, three factor matrices corresponding to users, items and time's representation as U, V and T separately, and the core tensor as $\mathcal{G}$. Specifically, N_u, N_v and N_T mean the number of users, items and time steps (each time step in my work equals one month), D_u, D_v is the length of users and items' representation (which depend on the given *side information*), and D_T is the length of each time step's representation. In consequence, we will have three-dimensional rating tensor $\mathcal{G} \in \mathbb{R}^{N_T \times N_u \times N_v}$, users' *side information* $X_u \in \mathbb{R}^{N_u \times D_u}$, items' *side information* $X_v \in \mathbb{R}^{N_v \times D_v}$, three factor matrices $U \in \mathbb{R}^{N_u \times D_u},\ V \in \mathbb{R}^{N_v \times D_v},\ T \in \mathbb{R}^{N_T \times D_T}$ as well as the core tensor $\mathcal{G} \in \mathbb{R}^{D_T \times D_u \times D_v}$.

2.2 Graph representation

The task of tensor completion consists of predicting the values of unobserved entries, which could be cast as a link prediction problem on a series of bipartite user-item graphs sharing the same nodes. In our problem setting, a particular bipartite user-item graph is denoted as $G_t = (\mathcal{W}_t, \mathcal{E}_t, \mathcal{R}_t), t = 1, 2, ..., N_T$ with entities consisting of a collection of user nodes $u_{t,i} \in \mathcal{W}_{t,u}$ with $i \in 1, ..., N_u$ and item nodes $v_{t,j} \in \mathcal{W}_{t,v}$ with $j \in 1, ..., N_v$, such that $\mathcal{W}_{t,u} \cup \mathcal{W}_{t,v} = \mathcal{W}_t$ and $\mathcal{W}_{t,u} \cup \mathcal{W}_{t,v} = \varnothing$. And we will have $\forall i \neq j,\ \mathcal{W}_i = \mathcal{W}_j = \mathcal{W},\ \mathcal{E}_i \neq \mathcal{E}_j, \mathcal{R}_i = \mathcal{R}_j = \mathcal{R} = \{1, 2, 3, 4, 5\}$. Specifically, the edges $(u_{t,i}, r_t, v_{t,j}) \in \mathcal{E}_t$ carry ratings (or labels if we consider

Table 2-1 Notations concerning tensor

notation	meaning
$\mathcal{X}$	three-dimensional rating tensor
U	user factor matrix obtained through tucker decomposition
V	item factor matrix obtained through tucker decomposition
T	time factor matrix obtained through tucker decomposition
$\mathcal{G}$	core tensor obtained through tucker decomposition
X_u	users' *side information*
X_v	items' *side information*
N_u	the number of users
N_v	the number of items
N_T	the number of discrete time steps
D_u	the length of each user's representation
D_v	the length of each item's representation
D_T	the length of each time step's representation
$\times_n$	tensor's n-mode product which will be introduced in section 3

predict ratings as a classification problem). This representation has been previously explored in [5] and leads to the development of graph-based methods in recommender systems.

To improve the quality of the representations by aggregating information from neighbor nodes, the graph convolutional network (GCN) [6] for single dimensional graphs is designed and has better performance than traditional matrix decomposition algorithms in terms of predicting missing entries. When it comes to a multi-dimensional graph, all the dimensions share the same set of nodes, whereas nodes don't only interact with those within the same graph, but also with those in different dimensions. As a result, it is likely for a given node to have different neighbors in different dimensions and I just call them *within-dimension neighbors* for nodes in each dimension. Apart from this, the same node in different dimension would also interact with each other inherently, and how to incorporate the across-dimension interactions is crucial to nodes' representation learning. Inspired by [2], I define the copies of nodes (within a particular dimension) in the other dimensions as their *across-dimension neighbors*. In a nutshell, a node in a given dimension will have both within-dimension and across-dimension neighbors, leading to two types of interactions.

Based on the proposed two types of interactions, I extend the message passing strategy to the multi-dimensional graph. To be more specific, *Temporal GCN* doesn't only propagate information among neighbor nodes within a single dimension, but also propagate among their copies in other dimensions. And the second interaction called inter-graph message passing is based on the concept - *virtual graph*, which I've mentioned above.

In my work, since making predictions on users' ratings depends on the users' historical rating behavior, I assume that a particular user' rating over an item will be only affected by his or her ratings in the past six months, meaning the size of time window for inter-graph message passing is set to 6. This assumption is in some correspondence with reality, and also greatly improves computation efficiency. It's also worth mentioning that I split the bipartite graph corresponding to a particular month into five bipartite graphs, since the ratings in this dataset range from 1 to 5. And I will obtain five rating tensors as well as multi-dimensional graphs $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4$ and $\mathcal{M}_5$ for each rating level over time, whose entries are denoted as either 1 or 0 depending on the original ratings. Therefore, each graph corresponding to a particular rating level is denoted as $M_{r,t}, r = 1, 2, 3, 4, 5,\ t = 1, ..., N_T$.

Table 2-2 Notations concerning graph representation

notation	meaning
$\mathcal{M}_r$	binary rating tensor corresponding to rating-level r
$M_{r,t}$	binary rating matrix at $t, t = 1, ..., N_T$ corresponding to rating-level r
L	the number of convolutional layers in GCN
$\mathcal{U}_r^{(l)}$	users' representation tensor after message passing (including both intra- and inter-graph message passing) of the lth layer in GCN, $l = 0, 1, ..., L$
$U_{r,t}^{(l)}$	users' representation matrix at t after message passing of the lth layer in GCN, $l = 0, 1, ..., L$
$\mathcal{V}_r^{(l)}$	items' representation tensor after message passing (includin oh intra- and inter-graph message passing) of the lth layer in GCN, $l = 0, 1, ..., L$
$V_{r,t}^{(l)}$	items' representation matrix at t after message passing of the lth layer in GCN, $l = 0, 1, ..., L$
$W_{r,u}^{l}$	rating level r's layer-specific trainable weight matrix for users' representation learning
$W_{r,v}^{l}$	rating level r's layer-specific trainable weight matrix for items' representation learning
$A_{r,t}$	the adjacency matrix of the virtual graph at a given point in time and rating-level

3 Preliminaries

3.1 N-mode product

Just like matrices, tensors could also be multiplied together, though the notations are much more complicated. The n-mode product is defined for a N-way tensor $\mathcal{X} \in \mathbb{R}^{I_1 \in I_1 \times I_2 \times ... \times I_N}$ with a matrix $U \in \mathbb{R}^{J \times I_n}$ and denoted as $\mathcal{X} \times_n U$, which will return a N-way tensor of size $I_1 \times I_2 \times ... \times I_{n-1} \times J \times I_{n+1} \times ... \times I_N$. For each element in the tensor, we will obtain

$$(\mathcal{X} \times_n U)_{i_1 \ldots i_{n-1} j i_{n+1} \ldots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \ldots i_N} u_{j i_n}. \tag{3-1}$$

Alternatively, we could regard the n-mode product as a type of transformation on mode-n fibers (vectors obtained by fixing all indices of $\mathcal{X}$ except for the nth dimension). The result will be a tensor of the same order as $\mathcal{X}$ in which the mode-n fibers are multiplied by the matrix U. As a result, the length of each dimension of $\mathcal{Y} = \mathcal{X} \times_n U$ will be the same as the length of the corresponding dimension of $\mathcal{X}$ except for the the nth dimension. The nth dimension now has length J, since the mode-n fibers of the tensor $\mathcal{Y}$ are the results of multiplying the mode-n fibers of $\mathcal{X}$ by the matrix $U \in \mathbb{R}^{J \times I_n}$. And this idea can be expressed mathematically as follows:

$$\mathcal{Y} = \mathcal{X} \times_n U \iff Y_{(n)} = U X_{(n)}. \tag{3-2}$$

In addition, a few facts of the n-mode product of a tensor with a matrix are in order[7]. For distinct modes in a series of multiplications, the order of the multiplication is irrelevant, and we will have

$$\mathcal{X} \times_n U \times_m V = \mathcal{X} \times_m V \times_n U \; (m \neq n). \tag{3-3}$$

When the modes are the same, then

$$\mathcal{X} \times_n U \times_n V = \mathcal{X} \times_n (VU). \tag{3-4}$$

For the equation above to make sense, the shape of two matrices $U \in \mathbb{R}^{J_u \times I_n}$ and $V \in \mathbb{R}^{J_v \times I_m}$ should meet the condition $I_m = J_u$.

If I replace the two-dimensional matrix $U \in \mathbb{R}^{J \times I_n}$ with a vector $v \in \mathbb{R}^{I_n}$, the result of the n-mode product of the tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times ... \times I_N}$ with the vector can be denoted as $\mathcal{X} \bar{\times}_n v$ of size $I_1 \times I_2 \times ... \times I_{n-1} \times I_{n+1} \times ... \times I_N$ [7]. Elementwise, we have

$$(\mathcal{X}\bar{\times}_n v)_{i_1\ldots i_{n-1}i_{n+1}\ldots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \ldots i_N} v_{i_n}. \tag{3-5}$$

The basic idea of the n-mode product of a tensor with a vector is to compute the inner product of the tensor's mode-n fibers with the vector. However, when it comes to the n-mode vector product, the order of multiplication matters since the order of the intermediate results also changes. Specifically,

$$\mathcal{X}\bar{\times}_m u\bar{\times}_n v = (\mathcal{X}\bar{\times}_m u)\bar{\times}_{n-1} v = (\mathcal{X}\bar{\times}_n v)\bar{\times}_m u \; (m < n). \tag{3-6}$$

3.2 Tucker decomposition

In this part, I will briefly introduce some details about the tucker decomposition, which was first introduced by Tucker in [8] and subsequently refined. And thus the tucker decomposition goes by lots of names, such as three-mode factor analysis, higher-order SVD (HOSVD) and N-mode PCA, etc.

The tucker decomposition could be considered as an extension of higher-order PCA, for it decomposes a tensor into a core tensor (denoted as $\mathcal{G}$) transformed by a matrix along its each mode. Take the three-way tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ as an example, we will have

$$\begin{aligned}\mathcal{X} &\approx \mathcal{G} \times_1 A \times_2 B \times_3 C \\ &= \sum_{p=1}^{P}\sum_{q=1}^{Q}\sum_{r=1}^{R} g_{pqr} a_p \otimes b_q \otimes c_r.\end{aligned} \tag{3-7}$$

Here the symbol $\otimes$ represents the outer product between vectors. Through the tucker decomposition mentioned above, we obtain three factor matrices $A \in \mathbb{R}^{I\times P}$, $B \in \mathbb{R}^{J\times Q}$ as well as $C \in \mathbb{R}^{K\times R}$, which can also be regarded as the principal components in each mode. And here P, Q and R are the number of components or columns in the factor matrices. Apparently, each entry of the core tensor $\mathcal{G} \in \mathbb{R}^{P\times Q\times R}$ represents the level of interaction between the components, which endows the tucker decomposition with greater power to express some ternary relationships than the CP decomposition. For more understandable, Figure 3-1 illustrates the tucker decomposition.

Based on the knowledge of the n-mode product of a tensor with a matrix, I could further obtain express each element of the tensor in the form of the outer product of three vectors as well as their ternary relationship's weight.

$$x_{ijk} \approx \sum_{p=1}^{P}\sum_{q=1}^{Q}\sum_{r=1}^{R} g_{pqr} a_{ip} b_{jq} c_{kr} \; i = 1, ..., I; j = 1, ..., J; k = 1, ..., K. \tag{3-8}$$

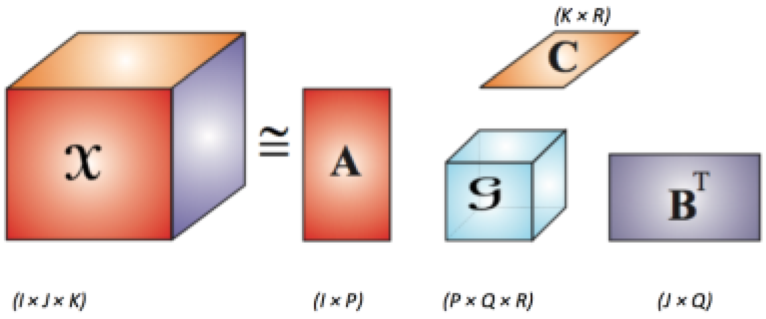

Figure 3-1 Tucker decomposition for a three-order tensor

From the perspective of data compression, when P, Q, R are smaller than I, J, K, we can compress the original tensor $\mathcal{X}$ using the tucker decomposition and obtain a compressed version the core tensor $\mathcal{G}$. The storage for the compressed core tensor $\mathcal{G}$ is significantly smaller than that for the tensor $\mathcal{X}$ in some cases. And we could easily extend the tucker decomposition to the N-order tensors as

$$\mathcal{X} \approx \mathcal{G} \times_1 A^{(1)} \times_2 A^{(2)} \cdots \times_N A^{(N)}. \tag{3-9}$$

Elementwise, we have

$$x_{i_1 i_2 \ldots i_N} \approx \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \cdots \sum_{r_N=1}^{R_N} g_{r_1 r_2 \ldots r_N} a^{(1)}_{i_1 r_1} a^{(2)}_{i_2 r_2} \ldots a^{(N)}_{i_N r_N}. \tag{3-10}$$

3.3 Graph convolutional networks (GCN)

In my study, I utilize graph convolutional networks (GCN) as a base component for making predictions on users' ratings over items in the future. I decide to employ this method because of its simplicity and effectiveness. In this section, I give a brief overview of GCN and will dwell on details of constructing the proposed *Temporal GCN* in the ensuing chapters.

As we know, graphs are widely applied in numerous domains, ranging from bioinformatics to social analysis. Graphs enable us to capture the underlying structural relations behind data and harvest more promising insights. In order to learn the representation of graphs in a low dimensional space, graph convolutional networks, a generalized version of the traditional CNN, has emerged and demonstrated its superior performance in many applications.

Generally speaking, there are mainly two types of graph convolutional networks based on ways of defining convolutions on graphs, which have been summarized in Figure 3-2. On the one hand, convolutions on graphs can be defined from the perspective of spectral. [9] gives us a comprehensive overview of graph signal processing, including some common operations on

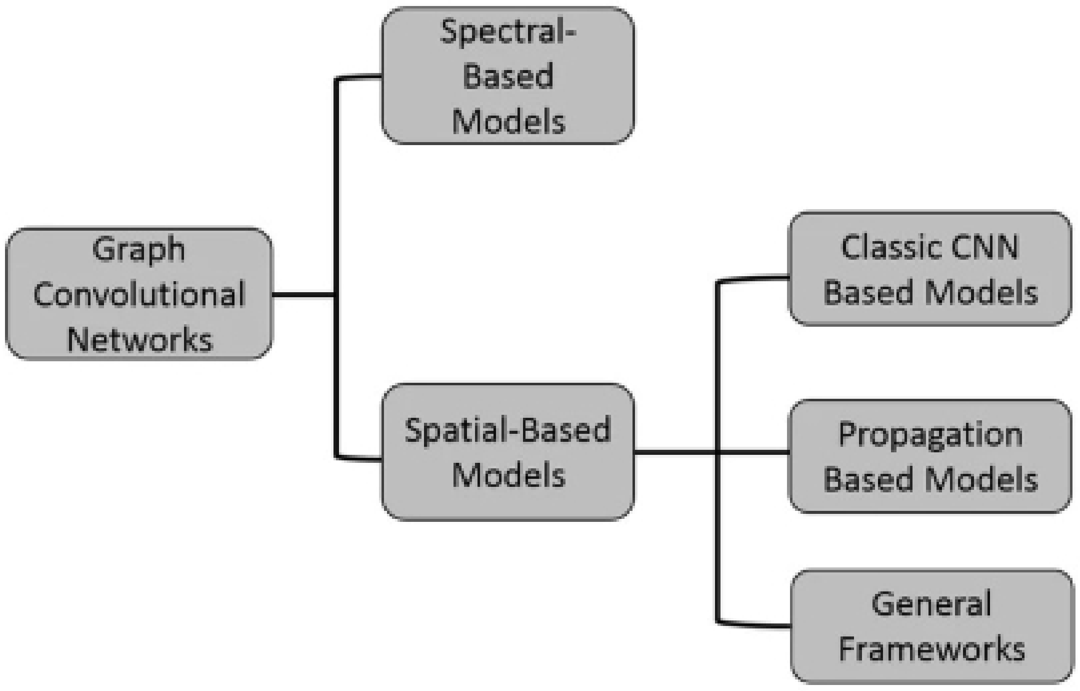

Figure 3-2 An overview of graph convolutional networks

graphs. In short, spectral graph convolutions are defined in the spectral domain on the basis of the Fourier transform on graphs. On the other hand, some researchers define convolutions in the spatial domain as the aggregations of neighborhoods' representations.

Since the spectral graph convolution depends on the specific sectral decomposition of Laplacian matrix, it's non-trivial to transfer this type of graph convolutional networks learned on one graph to another. That's also why I choose propagation-based spatial graph convolutional networks in this work. In the process of representation learning, hidden layers representations are computed by encoding both structures and nodes' properties with the message passing rule as follows,

$$H^{(l+1)} = f(H^{(l)}, A), l = 0, 1, \ldots, L. \tag{3-11}$$

$H^{(l)}$ is the feature matrix of the lth layer and A is the adjacency matrix of the graph. A widely used message passing rule is

$$H^{(l+1)} = f(H^{(l)}, A) = \sigma\left(\hat{A}H^{(l)}W^{(l)}\right), l = 0, 1, \ldots, L. \tag{3-12}$$

In the equation, $\hat{A} = \widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}$ is a normalization of the self-connected adjacency matrix $\widetilde{A} = A + I$, $\widetilde{D}$ is the diagonal degree matrix obtained from the adjacency matrix, and $W^{(l)}$ is a trainable weight matrix corresponding to lth layer in GCN. As for the non-linear activation function σ, we could use ReLU or Leaky ReLU to alleviate the vanishing gradient problem. However, σ is usually set as the softmax in the last layer. When $l = 0$, $H^{(0)}$ represents the initial feature matrix, whose each row corresponds to each node in the graph.

4 The proposed framework

4.1 Learning representations of points in time

In order to capture representations of each point in time, I employ the tucker decomposition, hoping to decompose the rating tensor as $\mathcal{X} \approx \mathcal{G} \times_1 T \times_2 U \times_3 V$ and learn the representations from the factor matrix corresponding to time. However, most tensor decomposition techniques, such as the CP-decomposition and tucker decomposition, have poor performances when given sparse tensors. One solution to this issue is to incorporate auxiliary information of users and items themselves.

Motivated by [10], I propose a constrained tucker decomposition, which incorporate users' and items' *side information* as the constraints on the factor matrices for users and items. In the mean time, these auxiliary information also serves as the regularization term, alleviating the over-fitting problem due to the sparsity issue. As a result, the modified tucker decomposition algorithm could be formulated as a constrained optimization problem. Firstly, using the Lagrange multiplier technique, I transform the constrained optimization problem into an unconstrained optimization problem. $\mathcal{W}$ is denoted as a tensor with missing observations and it has the same size as the rating tensor $\mathcal{X}$. Also the symbol $*$ here represents the element-wise product between two tensors of the same size.

The original constrained optimization problem is,

$$\begin{aligned} \min_{\mathcal{G},T,U,V} \quad & \|\mathcal{W} * (\mathcal{X} - \mathcal{G} \times_1 T \times_2 U \times_3 V)\|^2, \\ \text{s.t.} \quad & \|U - X_u\|_F^2 \leq \epsilon, \\ & \|V - X_v\|_F^2 \leq \gamma. \end{aligned} \tag{4-1}$$

And it could be transformed into an unconstrained optimization problem as follows,

$$\min_{\mathcal{G},T,U,V,\lambda,\mu} \quad \|\mathcal{W} * (\mathcal{X} - \mathcal{G} \times_1 T \times_2 U \times_3 V)\|^2 + \lambda\|U - X_u\|_F^2 + \mu\|V - X_v\|_F^2, \tag{4-2}$$

where λ and μ are Lagrange multipliers introduced to control the accuracy of approximation between the given *side information* and factor matrices obtained through the modified tucker decomposition. Just like I've mentioned above, *side information* also serves as a regularization technique to alleviate over-fitting problem by choosing different combinations of λ and μ.

For simplicity, here I denote the objective function of the unconstrained optimization problem as L. Then I compute the partial derivatives of L for the core tensor $\mathcal{G}$ and factor matrices T, U, V separately and apply gradient descent (GD), an iterative first-order optimization algorithm, to find the optimal solution. It is important to note as well that the gradient equations for the core tensor $\mathcal{G}$ and factor matrices T, U, V can be obtained through *matricization*. Specifically, matricization is the process of re-ordering elements of the tensor into a two-dimensional matrix. For more understandable, it could also be explained as unfolding or flattening a tensor, which is then transformed into a matrix.

The gradient equation of L for the core tensor $\mathcal{G}$ is as follows,

$$\frac{\partial L}{\partial \mathcal{G}} = -2\{\mathcal{W} * (\mathcal{X} - \mathcal{G} \times_1 T \times_2 U \times_3 V) \times_1 T^T \times_2 U^T \times_3 V^T\}. \tag{4-3}$$

The gradient equations of L for three factor matrices T, U, V are as follows separately,

$$\frac{\partial L}{\partial T} = -2\{[\mathcal{W} * (\mathcal{X} - \mathcal{G} \times_1 T \times_2 U \times_3 V)]_{(1)} \left[(\mathcal{G} \times_2 U \times_3 V)_{(1)}\right]^T\}, \tag{4-4}$$

$$\frac{\partial L}{\partial U} = -2\{[\mathcal{W} * (\mathcal{X} - \mathcal{G} \times_1 T \times_2 U \times_3 V)]_{(2)} \left[(\mathcal{G} \times_3 V \times_1 T)_{(2)}\right]^T\}, \tag{4-5}$$

$$\frac{\partial L}{\partial V} = -2\{[\mathcal{W} * (\mathcal{X} - \mathcal{G} \times_1 T \times_2 U \times_3 V)]_{(3)} \left[(\mathcal{G} \times_1 U \times_2 V)_{(3)}\right]^T\}. \tag{4-6}$$

After obtaining each trainable parameters' gradient equation, I further perform gradient descent to update all the parameters at the same time and continue this process until the objective function converges. The algorithm for learning representations of points in time is shown in Algorithm 1.

Having obtained representations of each point in time, I'm able to continue my research on modeling the inter-graph message passing strategy (or the across-dimension interactions in a multi-dimensional graph), which depends on different dimensions' representations and their importance to each other.

4.2 Constructing multi-dimensional *Temporal Graph*

In this section, I will discuss how to construct *Temporal Graph* using users' ratings over a period of time. The principle of constructing *Temporal Graph* is to cast matrix or tensor completion for recommender systems as link prediction on bipartite graphs. Given the accessibility of users and items' *side information*, representing rating matrices or tensors as bipartite graphs might help overcome the cold-start problem and have better performances than those matrix or tensor decompostion algorithms.

To begin with, the MovieLens dataset is composed of 6040 users' ratings over 3883 items

Algorithm 1 Tucker Decomposition with Side Information

```
procedure GD for Tucker Decomposition
    initialize $\mathcal{G} \in \mathbb{R}^{D_T \times D_u \times D_v}$, $T \in \mathbb{R}^{N_T \times D_T}$, $U \in \mathbb{R}^{N_u \times D_u}$, $V \in \mathbb{R}^{N_v \times D_v}$;
    repeat
        compute gradient for the core tensor $\mathcal{G}$ and three factor matrices $T, U, V$;
        $\mathcal{G} \leftarrow \mathcal{G} + \alpha\{\mathcal{W} * (\mathcal{X} - \mathcal{G} \times_1 T \times_2 U \times_3 V) \times_1 T^T \times_2 U^T \times_3 V^T\}$;
        $T \leftarrow T + \alpha\{[\mathcal{W} * (\mathcal{X} - \mathcal{G} \times_1 T \times_2 U \times_3 V)]_{(1)} \left[(\mathcal{G} \times_2 U \times_3 V)_{(1)}\right]^T\}$;
        $U \leftarrow U + \alpha\{[\mathcal{W} * (\mathcal{X} - \mathcal{G} \times_1 T \times_2 U \times_3 V)]_{(2)} \left[(\mathcal{G} \times_3 V \times_1 T)_{(2)}\right]^T\}$;
        $V \leftarrow V + \alpha\{[\mathcal{W} * (\mathcal{X} - \mathcal{G} \times_1 T \times_2 U \times_3 V)]_{(3)} \left[(\mathcal{G} \times_1 U \times_2 V)_{(3)}\right]^T\}$;
        compute the objective function $L$ using the updated parameters;
    until $L^{(t+1)} - L^{(t)} < \eta$
    return the core tensor $\mathcal{G}$ and factor matrices $T, U, V$.
end procedure
```

from to Feb 2000 to Jan 2002. In the original dataset, time for users' ratings is represented in the form of timestamp, and each users would have at least twenty ratings within the given period of time. For users, the dataset provides us their demographic information and only those who have provided their demographic information are included in the dataset. As for items or movies, MovieLens primarily contains their genres as well as their titles, which are identical to those provided by the IMDB. More details about the MovieLens dataset will be discussed in Section 5. I encode users and items' information and obtain two feature matrices for them. When it comes to the rating dataset, I transform the original relational table into the corresponding rating tensor $\mathcal{X}$, which only reserves the month of users' rating behavior.

After obtaining users' rating tensor $\mathcal{X}$, I further split it into five parts $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4$ and $\mathcal{M}_5$ for each rating level. Moreover, elements of original rating tensor are defined on $\{1, 2, 3, 4, 5\}$, whereas $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4$ and $\mathcal{M}_5$ are all binary rating tensors over time. These five binary tensors are proposed to construct multi-dimensional graphs and apply graph convolutional networks on them.

Take the binary rating tensor $\mathcal{M}_1$ as an example, $\mathcal{M}_1$ represents whether a user rate a specific movie as 1 star or not. For a binary rating matrix $M_{1,t} \in \mathbb{R}^{N_u \times N_v}$ at a specific point t in time, a nonzero entry $M_{1,t;ij}$ in the matrix represents an observed rating from a specific user for an item. $M_{1,t;ij}$ reflects an unobserved rating. Thus the binary rating matrix could be further cast as a bipartite graph, which consists of user nodes, item nodes as well as the entities denoting their *1 star* interactions.

To sum up, for a specific rating-level r, I could construct a series of bipartite graphs for each binary rating matrix at a specific point in time, which are then stacked one on top of another. As a result, I'm able to construct stacked bipartite *tensor graphs* over a period of time for five rating-levels separately.

4.3 Modeling the intra-graph interactions

In this section, I will introduce the process of modeling the inter-graph interactions. As I have mentioned above, the principle of spatial-based graph convolutional networks lies in the mechanism to coordinate heterogeneous information from each node's neighborhood and update their representation based on propagated information.

The intra-graph message passing is to harmonize information from neighbors within a graph. For instance, a user node u_i or a item node v_j in a given dimension d, I would perform the single dimensional graph convolutional networks on this dimension, only to obtain the structure and interactions within the dimension d. Since my work is based on users' rating datasets over time, I will replace d using r and t to represent a specific rating matrix corresponding to a given point in time and rating level. More specifically, the *intra-graph* learning process is almost the same as the message passing rule mentioned in Section 3.3, but there also exist subtle differences between them.

Most importantly, the graphs in my work are bipartite, which are made up of two independent sets of nodes and indices. So I need to modify the message passing rules and re-write them for users and items separately. For more understandable, Figure 4-1 gives an example of the bipartite graph in the field of recommendations.

For user nodes in the rating-r bipartite graph at t $(t = 1, \ldots, T)$,

$$U_{r,t}^{(l+1)} = f_u(U_{r,t}^{(l)}, M_{r,t}) = \sigma\left(M_{r,t}U_{r,t}^{(l)}W_{r,u}^{(l)}\right), l = 0, 1, \ldots, L. \tag{4-7}$$

In the equation, $M_{r,t}$ is a binary rating matrix corresponding to the rating-level r and the point in time t. $W_{r,u}^{(l)}$ is users' trainable weight matrix corresponding to lth layer in GCN as well as the rating level r. The non-linear activation function σ is often chosen to be ReLU or Leaky ReLU so as to alleviate the vanishing gradient problem. When $l = 0$, $U_{r,t}^{(0)}$ represents the initial feature matrix, whose each row corresponds to each user in the graph.

For item nodes in the rating-r bipartite graph at t $(t = 1, \ldots, T)$,

$$V_{r,t}^{(l+1)} = f_v(V_{r,t}^{(l)}, M_{r,t}) = \sigma\left(M_{r,t}^{T}V_{r,t}^{(l)}W_{r,v}^{(l)}\right), l = 0, 1, \ldots, L. \tag{4-8}$$

In the equation, $M_{r,t}$ is a binary rating matrix corresponding to the rating-level r and the point in time t. $W_{r,v}^{(l)}$ is items' trainable weight matrix corresponding to lth layer in GCN as well as the rating level r. The non-linear activation function σ is often chosen to be ReLU or Leaky ReLU so as to alleviate the vanishing gradient problem. When $l = 0$, $V_{r,t}^{(0)}$ represents the initial feature matrix, whose each row corresponds to each item in the graph.

Another significant difference between the intra-graph propagation in *Temporal GCN* and spatial-based graph convolution networks' message passing rule is that all graphs for each

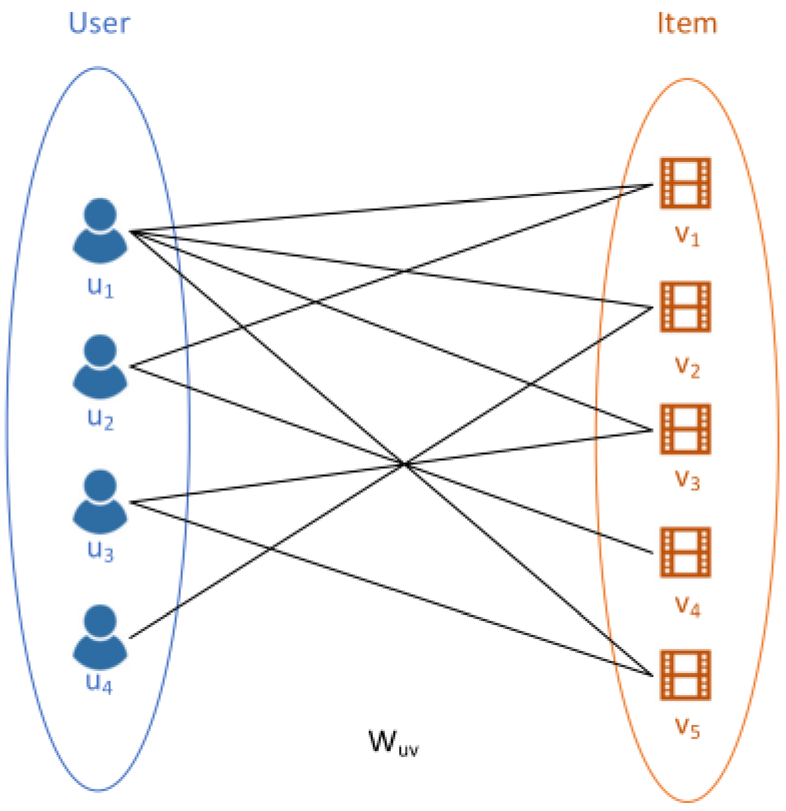

Figure 4-1 An example of the bipartite graph

rating level at different points in time have to perform the message passing mentioned above, leading to a tensor version.

4.4 Modeling the inter-graph interactions

Compared with modeling the intra-graph interactions, the idea behind the inter-graph propagation is a little complicated and highly relies on the concept called *virtual graph* mentioned in Section 2.

The inter-graph interactions are the basis of message passing between different graphs in the multi-dimensional graph, so that information of different points in time could be utilized and fused into an accordant one to make predictions. To achieve this goal, I connect with nodes' copies at other points in time and then construct virtual graphs. Moreover, virtual graphs take edges' direction and weight into account, since ratings in the future cannot affect ratings in the past. Also, each point's representation obtained by the tucker decomposition could help introduce an attention function $p_{g,d} = w_g^T M w_d$, measuring points' importance to each other. As a result, in order to perform the inter-graph message passing, I decide to take a weighted average over the representations of copies in the past six months (since the size of time window for inter-graph message passing is set to 6).

For user nodes $u_i,\ i = 7, \ldots, N_u$, its representation at t is the weighted average over the copies' representations and shown as follows,

$$u_{r,t} = \sum_{\tau=t-6}^{t-1} w_{t,\tau} u_{r,\tau}. \tag{4-9}$$

In this equation, $w_{t,\tau}$ is the weight for the point τ given the point t, modeling the importance of the point τ to the point t and satisfying $\sum\limits_{\tau=t-6}^{t-1} w_{t,\tau} = 1$. Naturally, different dimensions don't affect others equally. It is very likely that there exist pairs of similar dimensions, which are expected to contribute more when learning nodes' representations especially in the process of inter-graph message passing. Thanks to the factor matrix T returned by the tucker decomposition, each row of this matrix represents each point in time and is expected to contain some descriptive information of time. For instance, if two point in time are highly correlated, their representations ought be highly similar either. In consequence, I introduce an attention function to learn and model every two point in time's *similar score* based on the tucker decomposition's factor matrix T. To be more specific, the importance of a point in time g to another point d can be described as follows,

$$\begin{aligned} p_{g,d} &= att(t_g, t_d) \\ &= t_g^T M t_d. \end{aligned} \tag{4-10}$$

And I further apply a softmax function to guarantee the weight will satisfy the condition $\sum\limits_{\tau=t-6}^{t-1} w_{t,\tau} = 1$.

$$w_{g,d} = \frac{\exp(p_{g,d})}{\sum\limits_{g=1}^{D} \exp(p_{g,d})} \tag{4-11}$$

4.5 Predicting users' ratings in the future

In this section, I will discuss how to make full use of definitions and models mentioned above to make predictions on users' ratings in the near future.

The computation process of a layer of *Temporal GCN* is composed of an intra-graph as well as an inter-graph propagation. In this work, I set the number of *Temporal GCN* as 2 and conduct the intra- and inter-graph message passing just as I've introduced in the previous sections.

Here the tensors $\mathcal{U}_r^{(0)}(r = 1, \ldots, 5)$ and $\mathcal{V}_r^{(0)}(r = 1, \ldots, 5)$ are constructed from users and items' feature matrix. Each dimension of the tensors are just copies of the original feature matrices, which are then used for node message passing. After the inter-graph message passing at the first layer, users and items' representation matrix at a specific point in time for a given rating-level are denoted as $U^{(1)}_{r,t;intra}(r = 1, \dots, 5;\ t = 1, \dots, T)$ and $V^{(1)}_{r,t;intra}(r = 1, \dots, 5;\ t = 1, \dots, T)$. Based on the representations after the intra-graph message passing in the first layer of *Temporal GCN*, I continue to conduct the inter-graph message passing, and obtain their representations $U^{(1)}_{r,t;inter}(r = 1, \dots, 5;\ t = 1, \dots, T)$ as well as $V^{(1)}_{r,t;inter}(r = 1, \dots, 5;\ t = 1, \dots, T)$. The representation matrices can be computed as follows,

$$U^{(l+1)}_{r,t;intra} = \sigma\left(M_{r,t} U^{(l)}_{r,t} W^{(l)}_{r,u}\right), l = 0, 1, \tag{4-12}$$

$$V^{(l+1)}_{r,t;intra} = \sigma\left(M^T_{r,t} V^{(l)}_{r,t} W^{(l)}_{r,v}\right), l = 0, 1, \tag{4-13}$$

$$U^{(l+1)}_{r,t;inter} = \sigma\left(A_{r,t} U^{(l+1)}_{r,t;intra}\right), l = 0, 1, \tag{4-14}$$

$$V^{(l+1)}_{r,t;inter} = \sigma\left(A_{r,t} V^{(l+1)}_{r,t;intra}\right), l = 0, 1. \tag{4-15}$$

The adjacency matrices $A_{r,t}$ of the virtual graph is computed based on the attention function I mentioned in Section 4.4 and each of its element equals the importance of the point i to another point j.

After intra- and inter-graph message passing, I then stack the representation matrix $U^{(l+1)}_{r,t;intra}$ and $V^{(l+1)}_{r,t;intra}$ together and obtain the representation tensors $\mathcal{U}^{(l+1)}_r$ and $\mathcal{V}^{(l+1)}_r$. At the last layer of *Temporal GCN*, the framework will return the final representations of each user and item at different points $(t = 7, \dots, T)$ in time. In order to predict the ratings in the bipartite graphs, I employ a bilinear decoder, and regard each rating value as a separate class. Furthermore, the decoder will return a probability distribution over all rating values through the bilinear function as well as a softmax layer, which is as follows,

$$p(\hat{\mathcal{X}}_{t,i,j} = r) = \frac{\exp(U^T_{t-1,i} Q_r V_{t-1,j})}{\sum\limits_{k=1}^{5} \exp(U^T_{t-1,i} Q_k V_{t-1,j})}. \tag{4-16}$$

Here Q_r is another trainable matrix for each rating level, and the predicted rating is as follows,

$$\begin{aligned} \hat{\mathcal{X}}_{t,i,j} &= \sum_{r=1}^{5} r p(\hat{\mathcal{X}}_{t,i,j} = r) \\ &= \sum_{r=1}^{5} r \frac{\exp(U^T_{t-1,i} Q_r V_{t-1,j})}{\sum\limits_{k=1}^{5} \exp(U^T_{t-1,i} Q_k V_{t-1,j})}. \end{aligned} \tag{4-17}$$

4.6 Optimization and model training

To learn the parameters in the proposed framework, I need to construct a loss function as my optimization object, and regard the ratings after $t = 6$ as labels to predict. Specifically, I use ratings over every six months to make predictions on ratings in the next month, and compute the loss for the framework *Temporal GCN*. In order to construct my loss function, I utilize the method of the likelihood estimations of parameters and thus all I need to do is to minimize the function as follows,

$$\begin{aligned}\mathcal{L} &= - \sum_{t,i,j:\Omega_{t,i,j}=1} \sum_{r=1}^{5} \mathbb{I}\left[\mathcal{X}_{t,i,j} = r\right] \log\left(p(\hat{\mathcal{X}}_{t,i,j} = r)\right) \\ &= - \sum_{t,i,j:\Omega_{t,i,j}=1} \sum_{r=1}^{5} \mathbb{I}\left[\mathcal{X}_{t,i,j} = r\right] \log\left(\frac{\exp(U_{t-1,i}^{T} Q_r V_{t-1,j})}{\sum\limits_{k=1}^{5} \exp(U_{t-1,i}^{T} Q_k V_{t-1,j})}\right).\end{aligned} \tag{4-18}$$

Here $\mathbb{I}$ is a indicator function. The tensor $\Omega_{t,i,j} \in \{0,1\}^{N_u \times N_v \times N_T}$ is a mask for unobserved ratings in the original rating tensor $\mathcal{X}$, such that ones occur corresponding to observed entries in $\mathcal{X}$, and zeros for unobserved entries.

Apart from this, inspired by [1], I introduce mini-batching by sampling contributions to the objective function from different observed ratings, which means I only sample a fixed number of contributions from the sum over user, item and time pairs. This method could not only serve as a means of regularization, but also reduce the memory requirement to train the model. During the training process, I find it hard to read in the whole rating data sets and train the proposed model within the given time. In consequence, it is necessary to fit the full model for the MovieLens data sets into GPU memory. [1] experimentally verified that training with mini-batches and full batches leads to comparable results for the MovieLens data sets while adjusting for regularization parameters.

Table 4-1 Trainable parameters in *Temporal GCN*

notation	meaning
$V_{r,t}^{(l)}$	items' representation matrix at t after message passing of the lth layer in GCN,
$W_{r,u}^{l}$	rating level r's layer-specific trainable weight matrix for users' representation learning
$W_{r,v}^{l}$	rating level r's layer-specific trainable weight matrix for items' representation learning
M	trainable matrix introduced in the attention function
Q_r	trainable matrix introduced in the biliear decoder for each rating-level

In a nutshell, the parameters need to be learned are listed in Table 4-1.

5 Experiments

In this section, I demonstrate the effectiveness of the modified version of tucker decomposition with *side information* as well as the proposed framework *Temporal GCN* for predicting users' ratings in the future. I first introduce the rating data set in this work. Then, I will describe the tensor completion and predictions on users' ratings tasks together with discussions of the results.

5.1 Dataset

The files contain 1,000,209 anonymous ratings of about 3,900 movies from 6,040 MovieLens users who has joined MovieLens in 2000.

All ratings consist of *UserID*, *MovieID*, *Rating* and *Timestamp*. Specifically, *UserID*s range from 1 to 6040, and this data set includes 6040 users' *side information*. Also, each user will have at least twenty ratings within the given period. *MovieID*s range from 1 to 3952. However, some users' information is missing, and thus only 3883 movies' properties are available. $Rating$s are defined on a 5-star scale (only whole-star ratings are included), and *Timestamp* is shown in seconds.

User information contains their gender, age (pre-processed as an ordinal categorical variable) as well as occupation. To be more precise, occupation is chose from the choices listed in Table 5-1.

Movie information includes *MovieID*, *Title* as well as *Genre*. Specifically, *Title*s are identical to those provided by the IMDB (year of release are included as well). *Genre*s are pipe-separated and are selected from the genres listed in Table 5-2.

For computer processing of the *side information*, I encode users and items' information and obtain two feature matrices for them (each row of the matrices represents a user/movie's information). When it comes to the rating data set, I transform the original relational table into the corresponding rating tensor $\mathcal{X}$, which only reserves the month of users' rating behavior.

5.2 Tensor completion

In experiments, I randomly mask 20% observed entries, and use them as the test set to evaluate performances of several tensor completion algorithms. Also, I use the root mean squared error (RMSE) as the evaluation metric to compare different algorithms.

The modified tucker decomposition with *side information* completes the three-dimensional tensor. With the purpose of evaluating the performance of the tucker decomposi-

Table 5-1 Users' occupations

code	meaning
0	other or not specified
1	academic/educator
2	artist
3	clerical/admin
4	college/grad student
5	customer service
6	doctor/health care
7	executive/managerial
8	farmer
9	homemaker
10	K-12 student
11	lawyer
12	programmer
13	retired
14	sales/marketing
15	scientist
16	self-employed
17	technician/engineer
18	tradesman/craftsman
19	unemployed
20	writer

tion with side information, I also consider the tensor/matrix completion algorithms as follows,

- **Singular Value Decomposition (SVD)**: Singular value decomposition (SVD) is a matrix factorization technique. It is the generalization of the eigen-decomposition of a square matrix. In this work, I apply SVD on the rating matrix at each point in time, stack the reconstructed matrix together and compute the RMSE.
- **Tucker Decomposition (TD)**: The tucker decomposition (TD) decomposes the original three-dimensional rating tensor into three factor matrices T, U, V and a core tensor $\mathcal{G}$. This method simply apply the tucker decomposition without consideration of the *side information*.
- **Tucker Decomposition with *Side Information* (Side TD)**: The modified version of the tucker decomposition incorporates the *side information* of users and movies as the

Table 5-2 Movies' genres

genre	genre
Action	Film-Noir
Adventure	Horror
Animation	Musical
Children's	Mystery
Comedy	Romance
Crime	Sci-Fi
Documentary	Thriller
Drama	War
Fantasy	Western

constraints on the factor matrices U and V. With the assistance of the *side information*, this method could effectively alleviate the cold-start problem. And I will denote this method as **Side TD**.

The results of these matrix/tensor completion algorithms for MovieLens dataset are shown in Table 5-3. Here the hyper-parameters λ, μ as well as D_T of **Side TD** is set to $1000000, 800000$ and 5 separately.

Table 5-3 Performance comparison of tensor completion

method	dataset	RMSE
SVD	MovieLens	0.735
TD	MovieLens	0.0184619
Side TD	MovieLens + Feat	0.0102764

To be more specific, the process of tuning hyper-parameters for tucker decomposition with *side information* is summarized in Table 5-4 and Table 5-5. Hyper-parameters are optimized on an $80/20$ train/validation split of the original rating data set. As a result, I finally set λ to $100, 0000$, μ to $80, 0000$ and D_T to 5 in my work.

On the task of completing the rating tensor $\mathcal{X}$, the proposed tucker decomposition with *side information* of users and movies outperforms other methods for tensor completion. My results demonstrate the effectiveness of incorporating *side information* in rating tensor and learning representations along three dimensions (user, item and point in time). Moreover, the

Table 5-4 Tuning hyper-parameters for tucker decomposition (1)

$D_T = 6$	$\mu = 0$	$\mu = 100$	$\mu = 10000$	$\mu = 100000$	$\mu = 1000000$	$\mu = 10000000$
$\lambda = 0$	0.0184619	0.0184409	0.0183226	0.0159981	0.0163765	0.0157327
$\lambda = 100$	0.0184508	0.0183221	0.0180409	0.0147528	0.0153223	0.0154798
$\lambda = 10000$	0.0182472	0.0174796	0.0165821	0.0139214	0.0149843	0.0148683
$\lambda = 100000$	0.0161813	0.0159423	0.0145294	0.0124828	0.0128502	0.0132942
$\lambda = 1000000$	0.0149794	0.0145924	0.0117948	0.0109596	0.0112414	0.0120984
$\lambda = 10000000$	0.0157896	0.0155245	0.0137242	0.0129235	0.0133734	0.0142335

Table 5-5 Tuning hyper-parameters for tucker decomposition (2)

$D_T = 6$	$\mu = 200000$	$\mu = 400000$	$\mu = 600000$	$\mu = 800000$	$\mu = 1000000$
$\lambda = 200000$	0.0124224	0.0122466	0.0120232	0.0117252	0.0119386
$\lambda = 400000$	0.0123629	0.0120974	0.0117591	0.0114738	0.0116834
$\lambda = 600000$	0.0121895	0.0118367	0.0115694	0.0112942	0.0115238
$\lambda = 800000$	0.0118209	0.0116917	0.0114495	0.0112204	0.0114927
$\lambda = 1000000$	0.0115942	0.0114276	0.0111991	**0.0109596**	0.0110763

tucker decomposition has a better performance than singular value decomposition (SVD). This could explained by the three-dimensional tensor's capability of preserving the latent structure of time and modeling the interactions between user-item pair and time.

5.3 Predictions on users' ratings

The effectiveness of *TensorGCN* has been demonstareted in [3] compared with graph convolutional networks on a single graph. *TensorGCN* preserve and further exploit the information across different dimensions as well as their interactions. In this work, the proposed framework *Temporal GCN* is a variant of *TensorGCN*, which introduce an attention function to capture different dimensions' relationship and make full use of the representations of time learned from the tucker decomposition in the last subsection.

To validate the effectiveness of *Temporal GCN*, I decide to compare two methods to predict users' ratings as follows,

- **TensorGCN**: TensorGCN is a framework proposed in [3], which aims to learn nodes' presentations from a multi-dimensional graph. Specifically, a multi-dimensional graph consists of a series of graphs sharing the set of nodes with different set of edges in each dimension. The reason why it outperform traditional graph convolutional networks lies in its capability of preserving and modeling interactions across dimensions. However, it fails to consider relationships among different dimensions. Instead, when propagating nodes' message across dimensions, this framework treats each dimension equally, which does not comply with our common sense.

- **Temporal GCN**: Based on representations of each point in time obtained from the tucker decomposition as well as the attention function, I could better model interactions among dimensions and describe their similarity. To improve the computational efficiency, I assume that a particular users' rating over an item wizll be only affected by his or her ratings in the past six months. In my work, the number of layers for *Temporal GCN* equal 2, which achieves satisfactory results on the task of predicting users' future ratings.

The results of these two prediction methods for MovieLens dataset are shown in Table 5-6.

Table 5-6 Performance comparison of tensor completion

method	dataset	RMSE
TensorGCN (one layer)	MovieLens + Feat	1.004
TensorGCN (two layers)	MovieLens + Feat	0.996
Temporal GCN (one layer)	MovieLens + Feat	0.922
Temporal GCN (two layers)	MovieLens + Feat	0.910

On the task of predicting users' ratings, the proposed framework in this work *Temporal GCN* outperforms other methods for ratings' predictions. The results listed in Table 5-6 validate the effectiveness of introducing the attention mechanism and representations of points in time. Apart from this, for both *TensorGCN* and *Temporal GCN*, the increase of convolutional layers in the network might not contribute a lot to the accuracy of predicting users' ratings in the future.

6 Conclusions

In this work, I develop a modified tucker decomposition (which incorporate the *side information* as constraints on factor matrices for users and items) as well as a novel rating prediction framework named *Temporal GCN*.

Firstly, I propose to regard the *side information* as constraints on factor matrices returned by the tucker decomposition, aiming to incorporate users and items' auxiliary information and alleviate the over-fitting problem caused by the sparsity issue. And I further transform the constrained optimization problem into an unconstrained problem using the Lagrange multiplier technique, which could be solved through gradient descent method. I conduct comprehensive experiments on the MovieLens dataset, and validate the effectiveness of the tucker decomposition with the *side information* in recovering three-dimensional tensors and learning representations of points in time.

Secondly, I combine the multi-dimensional graph convolutional network framework and an attention mechanism to model interactions across dimensions. Specifically, I introduce a directed virtual graph to conduct nodes' message passing over time and use similarity between points in time as weight, which is based on the factor matrix returned by the tucker decomposition. Then I demonstrate the effectiveness of the proposed prediction framework *Temporal GCN* on the MovieLens dataset to make predictions on users' ratings in the next month.

References

[1] Berg R v d, Kipf T N, Welling M. Graph convolutional matrix completion[J]. arXiv preprint arXiv:1706.02263, 2017.

[2] Ma Y, Wang S, Aggarwal C C, et al. Multi-dimensional graph convolutional networks [C]//Proceedings of the 2019 siam international conference on data mining. SIAM, 2019: 657-665.

[3] Liu X, You X, Zhang X, et al. Tensor graph convolutional networks for text classification [C]//Proceedings of the AAAI conference on artificial intelligence: volume 34. 2020: 8409-8416.

[4] Harper F M, Konstan J A. The movielens datasets: History and context[J]. Acm transactions on interactive intelligent systems (tiis), 2015, 5(4):1-19.

[5] Li X, Chen H. Recommendation as link prediction in bipartite graphs: A graph kernel-based machine learning approach[J]. Decision Support Systems, 2013, 54(2):880-890.

[6] Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks [J]. arXiv preprint arXiv:1609.02907, 2016.

[7] Kolda T G, Bader B W. Tensor decompositions and applications[J]. SIAM review, 2009, 51(3):455-500.

[8] Tucker L R. Implications of factor analysis of three-way matrices for measurement of change[M]//Harris C W. Problems in measuring change. Madison WI: University of Wisconsin Press, 1963: 122-137.

[9] Shuman D I, Narang S K, Frossard P, et al. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains[J/OL]. IEEE Signal Processing Magazine, 2013, 30(3):83-98. DOI: 10.1109/MSP.2012.2235192.

[10] Chen Z, Gai S, Wang D. Deep tensor factorization for multi-criteria recommender systems [C]//2019 IEEE International Conference on Big Data (Big Data). IEEE, 2019: 1046-1051.

Appendix A Python codes

A.1 Data preprocessing

Code A-1 Data Preprocessing

```
import numpy as np
import pandas as pd
import os
import time
from datetime import datetime
from collections import defaultdict

filename = "pro_user.xlsx"
df = pd.read_excel(filename)
df["Age"] = (df["Age"] * 55).apply(lambda x: round(x)) + 1

type_list = [
    "Action", "Adventure", "Animation", "Children's", "Comedy", "Crime",
    "Documentary", "Drama", "Fantasy", "Film-Noir", "Horror", "Musical", "Mystery",
    "Romance", "Sci-Fi", "Thriller", "War", "Western"]

filename = "movies.dat"
f = open(filename, encoding="ISO-8859-1")

movie_info = []
for line in f:
    info = line.split("::")
    (movie_id, name, types) = info
    movie_id = int(movie_id)
    types = types.strip()
    # tmp = types.split("|")
    tmp_info = {"movie_id": movie_id, "name": name}
    for i in type_list:
        if i in types.split("|"):
            tmp_info[i] = 1
        else:
            tmp_info[i] = 0
    movie_info.append(tmp_info)

filename = "ratings.dat"
f = open(filename, encoding="ISO-8859-1")
time_list = []
for line in f:
    time_list.append(int(line.split("::")[-1]))

filename = "ratings.dat"
f = open(filename, encoding="ISO-8859-1")

rating_info = []
for line in f:
    info = line.split("::")
    (user_id, movie_id, rating, timestamp) = info
    user_id = int(user_id)
    movie_id = int(movie_id)
    rating = int(rating)
    timestamp = int(timestamp)
    year_ = datetime.fromtimestamp(timestamp).year - 2000
    month_ = datetime.fromtimestamp(timestamp).month - 4
    month_ = month_ + year_ * 12

    tmp_info = {"UserID": user_id, "MovieID": movie_id,
             "Rating": rating, "TimeStamp": timestamp,
             "Month": month_}
    rating_info.append(tmp_info)

TOTAL_MONTH = 36
BATCH_SIZE = 3
for i in range(TOTAL_MONTH // BATCH_SIZE):
    batch_rating_df = rating_df[(rating_df["Month"] >= (i*BATCH_SIZE)) & (rating_df["Month"] <
        ((i+1)*BATCH_SIZE))]
    a = pd.DataFrame({'UserID': user_list,
                    'key':[1 for _ in range(len(user_list))]})
    b = pd.DataFrame({"MovieID": movie_list,\
                    'key':[1 for _ in range(len(movie_list))]})
    c = pd.DataFrame({"Month": month_list[:BATCH_SIZE],
                    'key': [1 for _ in range(i*BATCH_SIZE, (i+1)*BATCH_SIZE)]})
    # batch上user,movie和time的笛卡尔积
    batch_dec = a.merge(b, on='key').merge(c, on="key").drop("key", axis=1)
    # 将两者合并，使用Nan对空缺的地方进行填充
    tmp_rating_df = batch_rating_df.merge(
          batch_dec, on=["UserID", "MovieID", "Month"],
          how="right").sort_values(["Month", "UserID", "MovieID"])

    tmp_rating_tensor = tmp_rating_df.groupby(["Month", "UserID"])["Rating"].apply(list)
    tmp_rating_tensor = np.array(list(tmp_rating_tensor))
    # reshape as (batch_NT, N_u, N_v)
    tmp_rating_tensor = tmp_rating_tensor.reshape(
          (BATCH_SIZE, (6040*BATCH_SIZE)//BATCH_SIZE, 3706))
    np.savetxt("data\\rating_tensor\\rating_tensor_%d.csv" % (i), item_matrix, delimiter=",")
```

A.2 Tucker Decomposition with Side Information

Code A-2 Saving Rating Dataset as Tensors

```
import numpy as np
import pandas as pd

user_df = pd.read_csv("data\\users.csv")
movie_df = pd.read_csv("data\\movies.csv")
rating_df = pd.read_csv("data\\ratings.csv").drop("TimeStamp", axis=1)

user_list = list(set(rating_df["UserID"]))
movie_list = list(set(rating_df["MovieID"]))
month_list = list(set(rating_df["Month"]))

TOTAL_MONTH = len(month_list)
a = pd.DataFrame({'UserID': user_list,
                'key':[1 for _ in range(len(user_list))]})
b = pd.DataFrame({"MovieID": movie_list,\
                'key':[1 for _ in range(len(movie_list))]})
c = pd.DataFrame({"Month": month_list[:TOTAL_MONTH],
                'key': [1 for _ in range(i*TOTAL_MONTH, (i+1)*TOTAL_MONTH)]})
# batch上user,movie和time的笛卡尔积
```

```python
batch_dec = a.merge(b, on='key').merge(c, on="key").drop("key", axis=1)

# 将两者合并，使用Nan对空缺的地方进行填充
tmp_rating_df = rating_df.merge(batch_dec,
                                on=["UserID", "MovieID", "Month"],
                                how="right").sort_values(["Month", "UserID", "MovieID"])

X = tmp_rating_df.groupby(["Month", "UserID"])["Rating"].apply(list)
X = np.array(list(X))
# reshape as (batch_NT, N_u, N_v)
X = X.reshape((TOTAL_MONTH, (6040*TOTAL_MONTH)//TOTAL_MONTH, 3706))

user_matrix = np.loadtxt("data\\user_matrix.csv", delimiter=",")
item_matrix = np.loadtxt("data\\item_matrix.csv", delimiter=",")
```

Code A-3   Tucker Decompostion with Side Information Optimized through Gradient Descent

```python
import numpy as np
import pandas as pd
from tensorly.tenalg import multi_mode_dot

class sidedTuckerDecomposition():
    """Methods for three-way tensor's Tucker Decompostion with users and items' side
        information.
    Parameters
    ----------
    max_iter: maximum number of iterations of Gradient Descent
    tol: tolerance for Gradient Descent in terms of change in reconstruction error
    """
    def __init__(self, max_iter=100, tol=1e-6):
        self.max_iter = max_iter
        self.tol = tol
        self.error = [np.inf, ]
        self.delta_error = np.inf
        self.iter = 0

    def matricization(self, tensor, axis):
        """Matricizing a tensor following the Kolda and Bader's definition along the given axis
        """
        return np.reshape(np.moveaxis(tensor, axis, 0),
                          (tensor.shape[axis], -1), order="F")

    def reconstructed_error(self, core_tensor, T, U, V, test=False):
        """Computing reconstructed error tensor through n-mode product using the core_tensor
            and
        three factor matrices T, U, V, which correspond to time, users and itemsseparately.
        """
        reconstructed_X = multi_mode_dot(tensor=core_tensor,
                                         matrix_or_vec_list=[T, U, V], modes=[0, 1, 2])
        if test:
            reconstructed_error = reconstructed_X - self.X_test
            reconstructed_error[np.isnan(self.X_test)] = 0
        else:
            reconstructed_error = reconstructed_X - self.X
            reconstructed_error[np.isnan(self.X)] = 0
        return reconstructed_error

    def tucker_output(self):
        print(f'Number of iterations: {self.iter}')
        print('Training reconstruction loss: {:.8f}'.format(self.error[-1]))
        return None

    def gradientDescent(self):
        """Performing gradient descent for the parameters to learn in Tucker Decomposition.
        """
        # initialize parameters (core_tensor, T, U, V) in Tucker Decompostion
        np.random.seed(0)
        self.core_tensor = np.random.rand(self.D_T, self.user_matrix.shape[1],
            self.item_matrix.shape[1])
        self.T = np.random.rand(self.X.shape[0], self.D_T)
        self.U = np.random.rand(self.X.shape[1], self.user_matrix.shape[1])
        self.V = np.random.rand(self.X.shape[2], self.item_matrix.shape[1])

        # perform gradient descent
        while self.delta_error >= self.tol and self.iter < self.max_iter:
            # compute gradient for each parameter
            error_tensor = self.reconstructed_error(self.core_tensor, self.T, self.U, self.V)
            dG = 2 * multi_mode_dot(tensor=error_tensor,
                                    matrix_or_vec_list=[self.T.T, self.U.T, self.V.T],
                                    modes=[0, 1, 2])
            dT = 2 * np.dot(self.matricization(error_tensor, 0),
                             self.matricization(multi_mode_dot(tensor=self.core_tensor,
                                              matrix_or_vec_list=[self.U, self.V],
                                              modes=[1, 2]), 0).T)
            dU = 2 * np.dot(self.matricization(error_tensor, 1),
                             self.matricization(multi_mode_dot(tensor=self.core_tensor,
                                              matrix_or_vec_list=[self.V, self.T],
                                              modes=[2, 0]), 1).T) + 2 * self.lambd * (self.U
                                                  self.user_matrix)
            dV = 2 * np.dot(self.matricization(error_tensor, 2),
                             self.matricization(multi_mode_dot(tensor=self.core_tensor,
                                              matrix_or_vec_list=[self.T, self.U],
                                              modes=[0, 1]), 2).T) + 2 *
                                                  self.item_matrix)

            self.error.append(np.sum(error_tensor ** 2) + \
                self.lambd * np.linalg.norm((self.U - self.user_matrix), "fro") + \
                self.mu                                              "fro"))
            self.delta_error = self.error[self.iter] - self.error[self.iter + 1]
            if self.train_verbose:
                self.tucker_output()
            # update parameters
            self.core_tensor -= self.eta * dG
            self.T -= self.eta * dT
            self.U -= self.eta * dU
            self.V -= self.eta * dV
            self.iter += 1

        self.final_error = self.error[-1]

    def fit(self, data, D_T, user_matrix, item_matrix,
            masked_ratio=0.2, lambd=1, mu=1, eta=0.03,
            train_verbose=False, test_verbose=False):
        """Methods to learn users, items and time's embeddings as well as their interaction.
        Parameters:
        ----------
        data: the original rating tensor, part of which will be masked, shape = (N_T, N_u, N_v)
        D_T: the number of features to present each time point
        user_matrix: users' side information, shape = (N_u, D_u)
        item_matrix: items' side information, shape = (N_v, D_v)
        masked_ratio: the ratio of masked observed values
        lambd, mu: two regularization hyperparameters, which correspond to users and items'
            side information
        eta: step for gradient descent
        train_verbose: boolean
        test_verbose: boolean
        """
        self.data = data
        # mask part of the data as the test set (split as X for train and X_test for test)
        X = self.data.copy().ravel()
        X_test = np.full_like(X, np.nan)
        observed_index = np.argwhere(~np.isnan(X))
        observed_num = len(observed_index)
        np.random.seed(0)
        masked_index = np.random.choice(observed_num, int(observed_num*masked_ratio),
            replace=False)
        masked_index = observed_index[masked_index]
        X_test[masked_index] = X[masked_index]
        X[masked_index] = np.nan
        self.X = X.reshape(data.shape)
        self.X_test = X_test.reshape(data.shape)

        self.D_T = D_T
        self.user_matrix = user_matrix
        self.item_matrix = item_matrix
        self.lambd = lambd
        self.mu = mu
        self.eta = eta
        self.train_verbose = train_verbose

        # perform gradient descent to learn the embeddings
        self.gradientDescent()

        # compute loss on the test set
        test_error_tensor = self.reconstructed_error(self.core_tensor, self.T, self.U, self.V,
            True)
        test_error = np.sum(test_error_tensor ** 2) / (~np.isnan(test_error_tensor)).sum()
        if test_verbose:
            if train_verbose:
                print('\nTest reconstruction error: {:.8f}'.format(test_error))
            else:
                print('Test reconstruction error: {:.8f}'.format(test_error))

        return test_error, self.core_tensor, self.T, self.U, self.V
```

A.3 Temporal GCN

Code A-4 Initialization

```
import numpy as np
import tensorflow as tf

def uniform(shape, scale=0.05, name=None):
    """Uniform init."""
    initial = tf.random_uniform(shape, minval=-scale, maxval=scale, dtype=tf.float32)
    return tf.Variable(initial, name=name)

def glorot(shape, name=None):
    """Glorot & Bengio (AISTATS 2010) init."""
    init_range = np.sqrt(6.0 / (shape[0] + shape[1]))
    initial = tf.random_uniform(shape, minval=-init_range, maxval=init_range, dtype=tf.float32)
    return tf.Variable(initial, name=name)
```

```python

def glorot_mix1(shape, name=None):
    """Glorot & Bengio (AISTATS 2010) init."""
    init_range = np.sqrt(6.0 / (shape[0] + shape[1]) / 1000)
    initial = tf.random_uniform(shape, minval=-init_range, maxval=init_range, dtype=tf.float32)
    return tf.Variable(initial, name=name)

def zeros(shape, name=None):
    """All zeros."""
    initial = tf.zeros(shape, dtype=tf.float32)
    return tf.Variable(initial, name=name)

def ones(shape, name=None):
    """All ones."""
    initial = tf.ones(shape, dtype=tf.float32)
    return tf.Variable(initial, name=name)
```

Code A-5 Constructing the model

```python
import tensorflow as tf
from inits import *
from utils import *

flags = tf.app.flags
FLAGS = flags.FLAGS

# global unique layer ID dictionary for layer name assignment
_LAYER_UIDS = {}

def get_layer_uid(layer_name=''):
    """Helper function, assigns unique layer IDs."""
    if layer_name not in _LAYER_UIDS:
        _LAYER_UIDS[layer_name] = 1
        return 1
    else:
        _LAYER_UIDS[layer_name] += 1
        return _LAYER_UIDS[layer_name]

def sparse_dropout(x, keep_prob, noise_shape):
    """Dropout for sparse tensors."""
    random_tensor = keep_prob
    random_tensor += tf.random_uniform(noise_shape)
    dropout_mask = tf.cast(tf.floor(random_tensor), dtype=tf.bool)
    pre_out = tf.sparse_retain(x, dropout_mask)
    return pre_out * (1. / keep_prob)

def dot(x, y, sparse=False):
    """Wrapper for tf.matmul (sparse vs dense)."""
    if sparse:
        res = tf.sparse_tensor_dense_matmul(x, y)
    else:
        res = tf.matmul(x, y)
    return res


class Layer(object):
    """Base layer class. Defines basic API for all layer objects.
    Implementation inspired by keras (http://keras.io).

    # Properties
        name: String, defines the variable scope of the layer.
        logging: Boolean, switches Tensorflow histogram logging on/off

    # Methods
        _call(inputs): Defines computation graph of layer
            (i.e. takes input, returns output)
        __call__(inputs): Wrapper for _call()
        _log_vars(): Log all variables
    """

    def __init__(self, **kwargs):
        allowed_kwargs = {'name', 'logging'}
        for kwarg in kwargs.keys():
            assert kwarg in allowed_kwargs, 'Invalid keyword argument: ' + kwarg
        name = kwargs.get('name')
        if not name:
            layer = self.__class__.__name__.lower()
            name = layer + '_' + str(get_layer_uid(layer))
        self.name = name
        self.vars = {}
        self.vars_mix = {}
        logging = kwargs.get('logging', False)
        self.logging = logging
        self.sparse_inputs = False

    def _call(self, inputs):
        return inputs

    def __call__(self, inputs):
        with tf.name_scope(self.name):
            if self.logging and not self.sparse_inputs:
                tf.summary.histogram(self.name + '/inputs', inputs)
            outputs = self._call(inputs)
            if self.logging:
                tf.summary.histogram(self.name + '/outputs', outputs)
            return outputs

    def _log_vars(self):
        for var in self.vars:
            tf.summary.histogram(self.name + '/vars/' + var, self.vars[var])

class GraphConvolution_mix1(Layer):
    """Graph convolution layer."""

    def __init__(self, input_dim, output_dim, placeholders, dropout=0.,
                 sparse_inputs=False, act=tf.nn.leaky_relu, bias=False,
                 featureless=False, **kwargs):
        super(GraphConvolution_mix1, self).__init__(**kwargs)

        if dropout:
            self.dropout = placeholders['dropout']
        else:
            self.dropout = 0.

        self.act = act
        self.support = placeholders['support']
        self.support_mix = placeholders['support_mix']
        self.sparse_inputs = sparse_inputs
        self.featureless = featureless
        self.bias = bias
        # helper variable for sparse dropout
        self.num_features_nonzero = placeholders['num_features_nonzero']
        self.input_dim = input_dim
        self.output_dim = output_dim
        with tf.variable_scope(self.name + '_vars'):
            for i in range(len(self.support_mix)):
                self.vars['weights_' + str(i)] = glorot([input_dim, output_dim],
                                                        name='weights_' + str(i))
                self.vars['weights_' + str(i) + str(i)] = glorot([output_dim, output_dim],
                                                                 name='weights_' + str(i) + str(i))
            if self.bias:
                self.vars['bias'] = zeros([output_dim], name='bias')

        if self.logging:
            self._log_vars()

    def atten(self, supports):
        for i in range(len(supports)):
            supports[i] = dot(supports[i], self.vars['weights_' + str(i) + str(i)])
        att_features = []
        att_features.append(tf.nn.leaky_relu(tf.add(supports[1], supports[2])))
        att_features.append(tf.nn.leaky_relu(tf.add(supports[0], supports[2])))
        att_features.append(tf.nn.leaky_relu(tf.add(supports[0], supports[1])))
        return att_features

    def _call(self, inputs):
        xx = inputs
        # dropout
        for i in range(len(xx)):
            if self.sparse_inputs:
                xx[i] = sparse_dropout(xx[i], 1 - self.dropout, self.num_features_nonzero)
            else:
                xx[i] = tf.nn.dropout(xx[i], 1 - self.dropout)

        # convolve
        supports = list()
        for i in range(len(self.support_mix)):
            if not self.featureless:
                pre_sup = dot(xx[i], self.vars['weights_' + str(i)],
                              sparse=self.sparse_inputs)
            else:
                pre_sup = self.vars['weights_' + str(i)]
            support = dot(self.support_mix[i], pre_sup, sparse=True)
            support = tf.nn.leaky_relu(support)
            supports.append(support)

        supports = self.atten(supports)

        # get embedding
        self.embedding = tf.stack([supports[-3], supports[-2], supports[-1]], axis=0)
        self.embedding = tf.reduce_mean(self.embedding, axis=0)
        return supports

class Model(object):
    def __init__(self, **kwargs):
        allowed_kwargs = {'name', 'logging'}
        for kwarg in kwargs.keys():
            assert kwarg in allowed_kwargs, 'Invalid keyword argument: ' + kwarg
        name = kwargs.get('name')
        if not name:
            name = self.__class__.__name__.lower()
        self.name = name

        logging = kwargs.get('logging', False)
        self.logging = logging
        self.vars = {}
        self.placeholders = {}

        self.layers = []
        self.activations = []

        self.inputs = None
        self.outputs = None

        self.loss = 0
        self.accuracy = 0
        self.optimizer = None
        self.opt_op = None

    def _build(self):
        raise NotImplementedError

    def build(self):
        """ Wrapper for _build() """
        with tf.variable_scope(self.name):
            self._build()

        # Build sequential layer model
        self.activations.append(self.inputs)
        self.activations.append(self.inputs)
        self.activations.append(self.inputs)
        for layer in self.layers:
            hidden = layer(self.activations[-3:])
            self.activations.extend(hidden)
        self.outputs = tf.stack([self.activations[-3], self.activations[-2],
         self.activations[-1]], axis=0)
        self.outputs = tf.reduce_mean(self.outputs, axis=0)
        variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope=self.name)

        self.vars = {var.name: var for var in variables}

        # Build metrics
        self._loss()
        self._accuracy()

        self.opt_op = self.optimizer.minimize(self.loss)

    def predict(self):
        pass

    def _loss(self):
        raise NotImplementedError

    def _accuracy(self):
        raise NotImplementedError

    def save(self, sess=None):
        if not sess:
            raise AttributeError("TensorFlow session not provided.")
        saver = tf.train.Saver(self.vars)
        save_path = saver.save(sess,
                               "../data_tgcn/mr/build_train/{}_best_models/%s.ckpt".format(FLAGS.dataset)
                                   % self.name)
        print("Model saved in file: %s" % save_path)

    def load(self, sess=None):
        if not sess:
            raise AttributeError("TensorFlow session not provided.")
        saver = tf.train.Saver(self.vars)
        save_path = "../data_tgcn/mr/build_train/{}_best_models/%s.ckpt".format(FLAGS.dataset)
            % self.name
        saver.restore(sess, save_path)
        print("Model restored from file: %s" % save_path)

class GCN(Model):
    def __init__(self, placeholders, input_dim, **kwargs):
        super(GCN, self).__init__(**kwargs)

        self.inputs = placeholders['features']
        self.input_dim = input_dim
        # self.input_dim = self.inputs.get_shape().as_list()[1]
        self.output_dim = placeholders['labels'].get_shape().as_list()[1]
        self.placeholders = placeholders
        self.optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.learning_rate)
        self.build

    def _loss(self):
        # Weight decay loss
        for var in self.layers[0].vars.values():
            self.loss += FLAGS.weight_decay * tf.nn.l2_loss(var)

        # Cross entropy error
        self.loss += masked_softmax_cross_entropy(self.outputs, self.placeholders['labels'],
                                                  self.placeholders['labels_mask'])

    def _accuracy(self):
        self.accuracy = masked_accuracy(self.outputs, self.placeholders['labels'],
                                        self.placeholders['labels_mask'])
        self.pred = tf.argmax(self.outputs, 1)
        self.labels = tf.argmax(self.placeholders['labels'], 1)

    def _build(self):
        self.layers.append(GraphConvolution_mix1(input_dim=self.input_dim,
                                                 output_dim=FLAGS.hidden1,
                                                 placeholders=self.placeholders,
                                                 act=tf.nn.leaky_relu,
                                                 dropout=True,
                                                 featureless=False,
                                                 sparse_inputs=True,
                                                 logging=self.logging))

        self.layers.append(GraphConvolution_mix1(input_dim=FLAGS.hidden1,
                                                 output_dim=self.output_dim,
                                                 placeholders=self.placeholders,
                                                 act=lambda x: x, #
                                                 dropout=True,
                                                 featureless=False,
                                                 sparse_inputs=False,
                                                 logging=self.logging))

    def predict(self):
        return tf.nn.softmax(self.outputs)
```