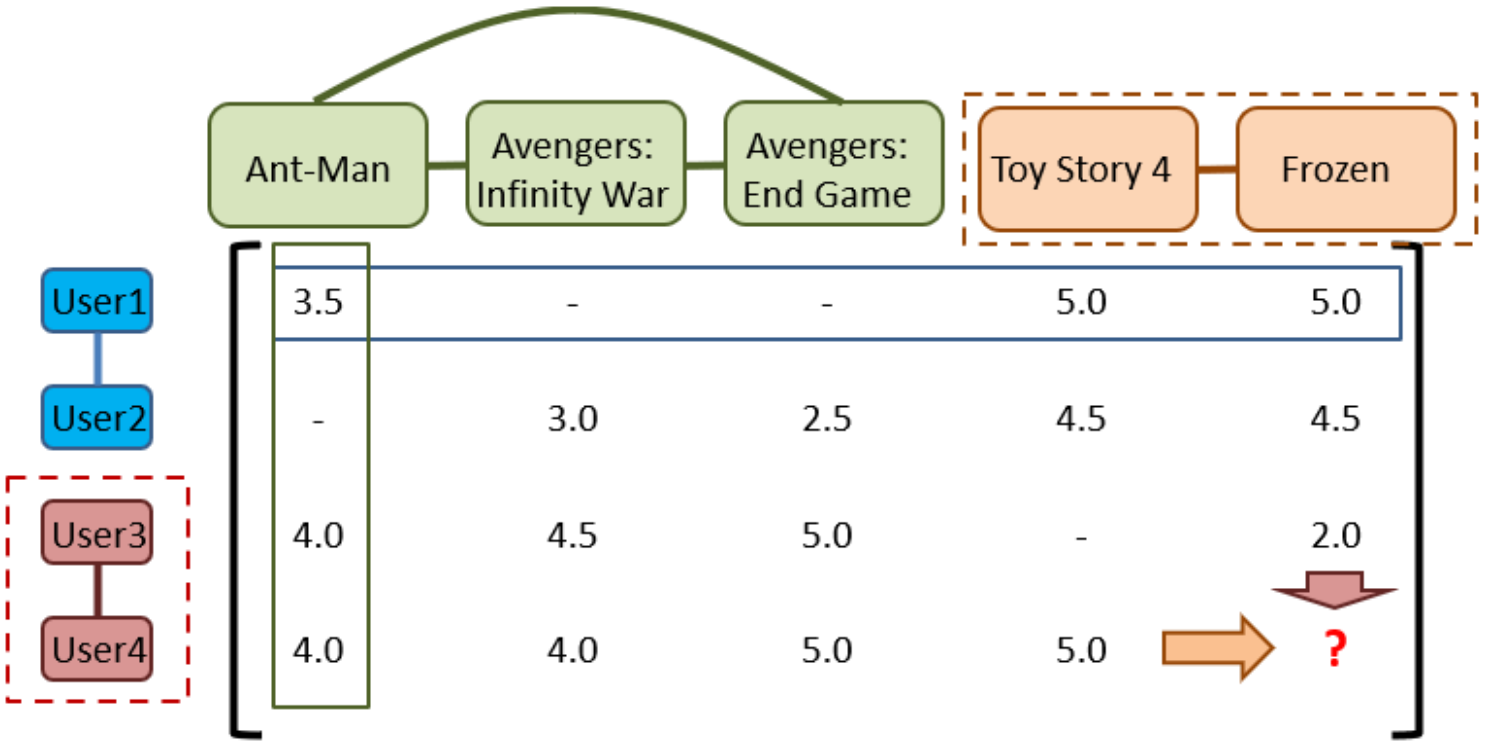
Geometric Matrix Completion with Recurrent Multi-Graph Neural Networks

Federico Monti, Michael M. Bronstein, and Xavier Bresson

NIPS 2017

Presenter : Mineui Hong

Recommendation & Matrix Completion



Geometric Matrix Completion

$$\min_X \underbrace{\|X\|_{g_r}^2 + \|X\|_{g_c}^2}_{\text{Smoothness over row/column-graph}} + \underbrace{\frac{\mu}{2}\|\Omega \circ (X - Y)\|_F^2}_{\text{Error on labeled data}}$$

$\|X\|_{g_r}^2 = trace(X^T \Delta_r X)$

→ **Multi-Graph Convolution**

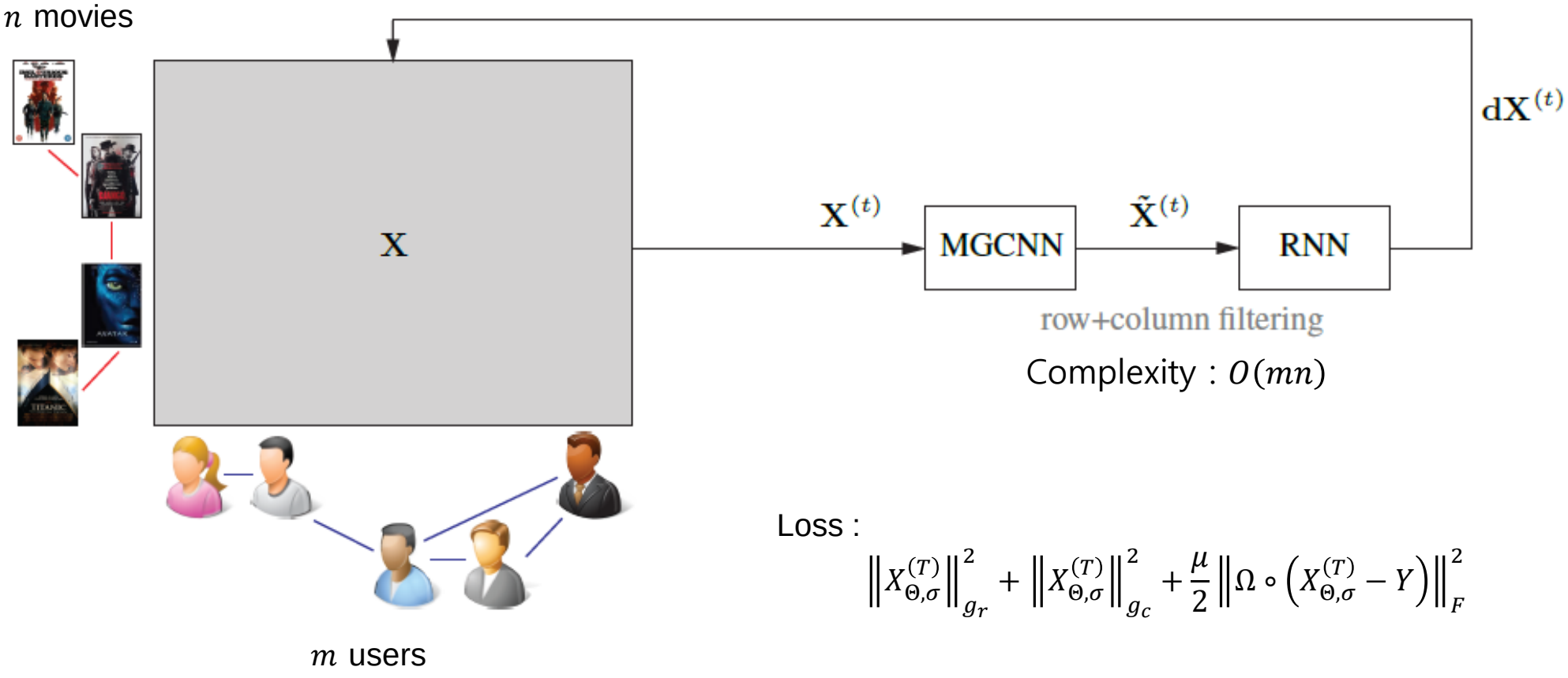
Multi-Graph Convolutional Neural Network (MGCNN)

	1-Dimensional	2-Dimensional
Graph Laplacian	$\Delta = \Phi\Lambda\Phi^{T}$ $\Lambda = \mathrm{diag}(\lambda_1, \cdots, \lambda_n)$ Φ : eigenvector matrix	$\Delta_c = \Phi_c\Lambda_c\Phi_c^{T}$, $\Delta_r = \Phi_r\Lambda_r\Phi_r^{T}$
Fourier Transform	$\hat{X} = \Phi^{T}X$	$\hat{X} = \Phi_r^{T}X\Phi_c$
Convolution Operation	$X * Y = \Phi(\Phi^{T}X) \circ (\Phi^{T}Y)$ $= \Phi(\hat{X} \circ \hat{Y})$	$X * Y = \Phi_r(\hat{X} \circ \hat{Y})\Phi_c^{T}$
Chebychev Polynomial Filter	$\tau_\theta(\lambda) = \sum_{j=0}^{p} \theta_j T_j(\tilde{\lambda})$, $\tilde{\lambda} \in [-1,1]$	$\tau_\Theta(\lambda_c, \lambda_r) = \sum_{j,j'=0}^{p} \theta_{j,j'} T_j(\tilde{\lambda}_c)\, T_{j'}(\tilde{\lambda}_r)$
Graph CNN	$\tilde{X}_l = \xi\left(\sum_{l'=1}^{q'}\sum_{j=0}^{p} \theta_{j,ll'} T_j(\tilde{\Delta})X_l\right)$ l', l : input / output channel ξ : nonlinearity function	$\tilde{X}_l = \xi\left(\sum_{l'=1}^{q'}\sum_{j,j'=0}^{p} \theta_{jj',ll'} T_j(\tilde{\Delta}_r)X_{l'} T_{j'}(\tilde{\Delta}_c)\right)$

Recurrent MGCNN (RMGCNN)

Matrix Diffusion : Make small changes for each step

$$\mathbf{X}^{(t+1)} = \mathbf{X}^{(t)} + \mathrm{d}\mathbf{X}^{(t)}$$



Loss :

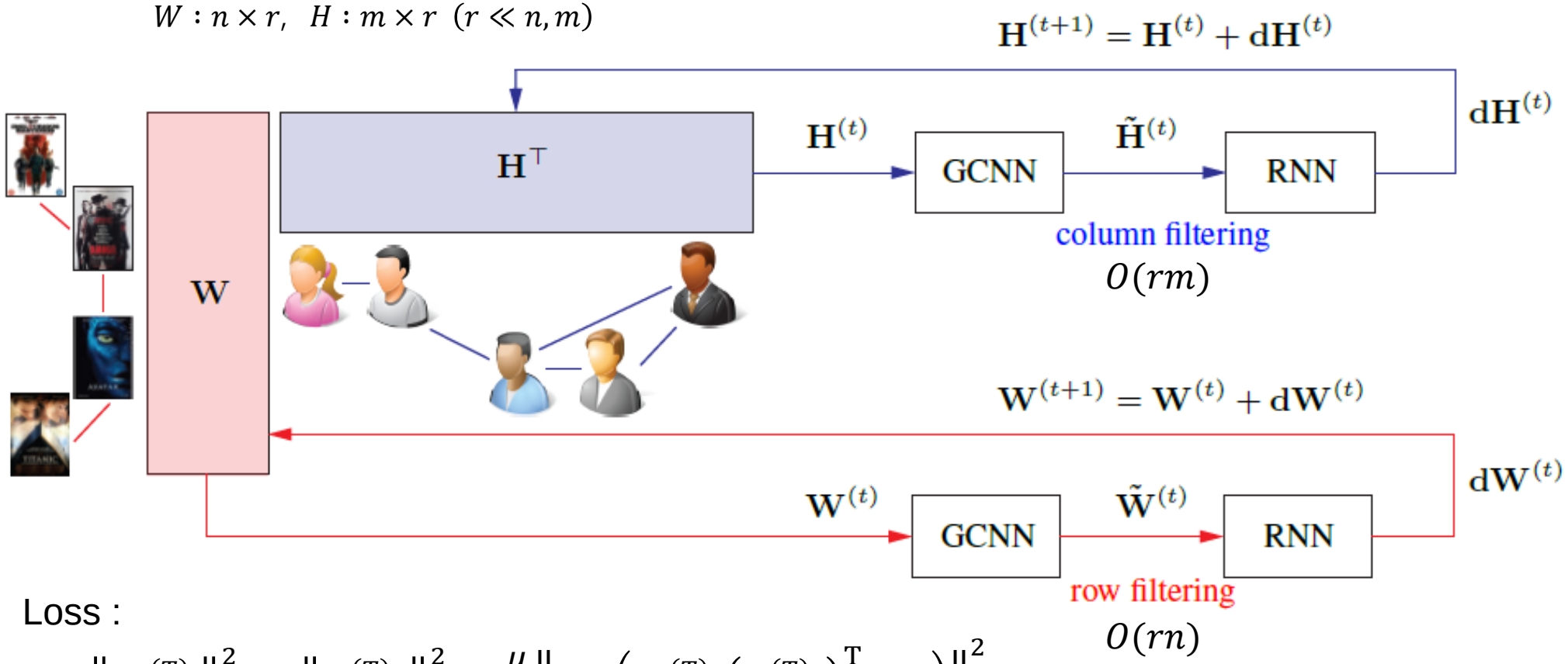
$$\left\|X^{(T)}_{\Theta,\sigma}\right\|^2_{g_r} + \left\|X^{(T)}_{\Theta,\sigma}\right\|^2_{g_c} + \frac{\mu}{2}\left\|\Omega \circ \left(X^{(T)}_{\Theta,\sigma} - Y\right)\right\|^2_F$$

Separable RMGCNN (sRMGCNN)

Total Learning Complexity : $O(n+m)$

Matrix Factorization

$$X = WH^T : n \times m$$
$$W : n \times r, \quad H : m \times r \quad (r \ll n, m)$$

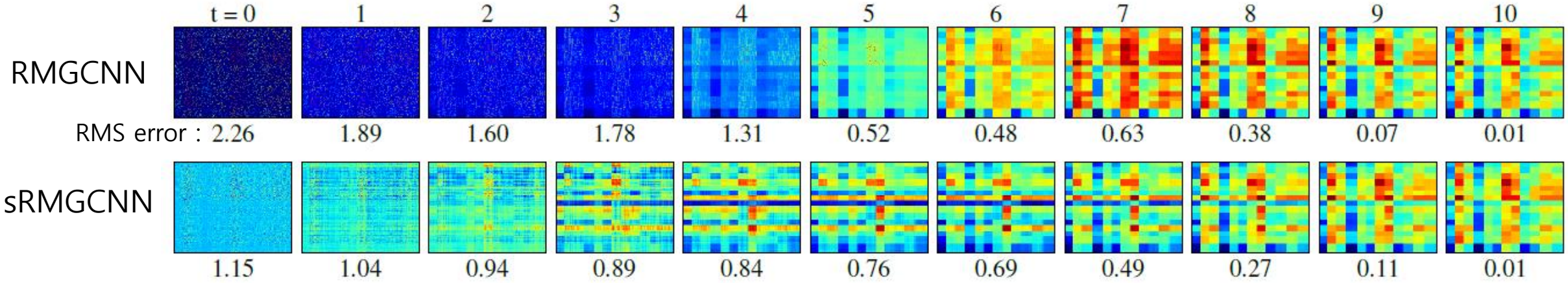


Loss :

$$\left\|W^{(T)}_{\Theta_r,\sigma}\right\|^2_{g_r} + \left\|H^{(T)}_{\Theta_c,\sigma}\right\|^2_{g_c} + \frac{\mu}{2}\left\|\Omega \circ \left(W^{(T)}_{\Theta_r,\sigma}\left(H^{(T)}_{\Theta_c,\sigma}\right)^{\mathrm{T}} - Y\right)\right\|^2_F$$

Experimental Results

Evolution of matrix $X^{(t)}$



Method	Params	Architecture	RMSE
$\text{MGCNN}_{\text{3layers}}$	$9K$	1MGC32, 32MGC10, 10MGC1	0.0116
$\text{MGCNN}_{\text{4layers}}$	$53K$	1MGC32, 32MGC32 $\times$ 2, 32MGC1	0.0073
$\text{MGCNN}_{\text{5layers}}$	$78K$	1MGC32, 32MGC32 $\times$ 3, 32MGC1	0.0074
$\text{MGCNN}_{\text{6layers}}$	$104K$	1MGC32, 32MGC32 $\times$ 4, 32MGC1	0.0064
RMGCNN	**9K**	1MGC32 + LSTM	**0.0053**

Experimental Results

METHOD	RMSE
GLOBAL MEAN	1.154
USER MEAN	1.063
MOVIE MEAN	1.033
MC [9]	0.973
IMC [17, 42]	1.653
GMC [19]	0.996
GRALS [33]	0.945
sRMGCNN	**0.929**

MovieLens dataset

METHOD	FLIXSTER	DOUBAN	YAHOOMUSIC
GRALS	1.3126 / 1.2447	0.8326	38.0423
sRMGCNN	**1.1788 / 0.9258**	**0.8012**	**22.4149**

Other dataset

Reproduction

Code is based on the authors' implementation (https://github.com/fmonti/mgcnn/), and modified for running it on Python3.6 & trying simplified Chebyshev filter.

Modified Code : https://github.com/mini-hong/sRMGCNN

Tested Environment :

- Python 3.6
- Tensorflow 1.13.1
- Other Requirements : joblib, h5py, scipy, matplotlib

Code explanation

Matrix of the given data & user/item graph weights are loaded from dataset

```
#loading of the required matrices
M = load_matlab_file(path_dataset, 'M')
Otraining = load_matlab_file(path_dataset, 'Otraining')
Otest = load_matlab_file(path_dataset, 'Otest')
Wrow = load_matlab_file(path_dataset, 'W_users') #sparse
Wcol = load_matlab_file(path_dataset, 'W_movies') #sparse
```

Construct graph Laplacians of user/item graph from the loaded data

```
#computation of the normalized laplacians
Lrow = sp.csgraph.laplacian(Wrow, normed=True)
Lcol = sp.csgraph.laplacian(Wcol, normed=True)
```

```
#apply SVD initially for detecting the main components of our initialization
U, s, V = np.linalg.svd(Odata*M, full_matrices=0)

print(U.shape)
print(s.shape)
print(V.shape)
```

```
(943, 943)
(943,)
(943, 1682)
```

Apply SVD and do **matrix factorization** by selecting top 10 rank of the original matrix

```
rank_W_H = 10
partial_s = s[:rank_W_H]
partial_S_sqrt = np.diag(np.sqrt(partial_s))
initial_W = np.dot(U[:, :rank_W_H], partial_S_sqrt)
initial_H = np.dot(partial_S_sqrt, V[:rank_W_H, :]).T

print(initial_W.shape)
print(initial_H.shape)
```

```
(943, 10)
(1682, 10)
```

Code explanation

```
##################################definition of the NN variables#####################################

#definition of the weights for extracting the global features
self.W_conv_W = tf.get_variable("W_conv_W", shape=[self.ord_col*initial_W.shape[1], self.n_conv_feat],
                                initializer=tf.contrib.layers.xavier_initializer())
self.b_conv_W = tf.Variable(tf.zeros([self.n_conv_feat,]))
self.W_conv_H = tf.get_variable("W_conv_H", shape=[self.ord_row*initial_W.shape[1], self.n_conv_feat],
                                initializer=tf.contrib.layers.xavier_initializer())
self.b_conv_H = tf.Variable(tf.zeros([self.n_conv_feat,]))

#recurrent N parameters
self.W_f_u = tf.get_variable("W_f_u", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.W_i_u = tf.get_variable("W_i_u", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.W_o_u = tf.get_variable("W_o_u", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.W_c_u = tf.get_variable("W_c_u", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.U_f_u = tf.get_variable("U_f_u", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.U_i_u = tf.get_variable("U_i_u", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.U_o_u = tf.get_variable("U_o_u", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.U_c_u = tf.get_variable("U_c_u", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.b_f_u = tf.Variable(tf.zeros([self.n_conv_feat,]))
self.b_i_u = tf.Variable(tf.zeros([self.n_conv_feat,]))
self.b_o_u = tf.Variable(tf.zeros([self.n_conv_feat,]))
self.b_c_u = tf.Variable(tf.zeros([self.n_conv_feat,]))

self.W_f_m = tf.get_variable("W_f_m", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.W_i_m = tf.get_variable("W_i_m", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.W_o_m = tf.get_variable("W_o_m", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.W_c_m = tf.get_variable("W_c_m", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.U_f_m = tf.get_variable("U_f_m", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.U_i_m = tf.get_variable("U_i_m", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.U_o_m = tf.get_variable("U_o_m", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.U_c_m = tf.get_variable("U_c_m", shape=[self.n_conv_feat, self.n_conv_feat],
                             initializer=tf.contrib.layers.xavier_initializer())
self.b_f_m = tf.Variable(tf.zeros([self.n_conv_feat,]))
self.b_i_m = tf.Variable(tf.zeros([self.n_conv_feat,]))
self.b_o_m = tf.Variable(tf.zeros([self.n_conv_feat,]))
self.b_c_m = tf.Variable(tf.zeros([self.n_conv_feat,]))
```

```
#output parameters
self.W_out_W = tf.get_variable("W_out_W", shape=[self.n_conv_feat, initial_W.shape[1]],
                               initializer=tf.contrib.layers.xavier_initializer())
self.b_out_W = tf.Variable(tf.zeros([initial_W.shape[1],]))
self.W_out_H = tf.get_variable("W_out_H", shape=[self.n_conv_feat, initial_H.shape[1]],
                               initializer=tf.contrib.layers.xavier_initializer())
self.b_out_H = tf.Variable(tf.zeros([initial_H.shape[1],]))

#########definition of the NN
#definition of W and H
self.W = tf.constant(initial_W.astype('float32'))
self.H = tf.constant(initial_H.astype('float32'))

self.X = tf.matmul(self.W, self.H, transpose_b=True) #we may initialize it at random here
self.list_X = list()
self.list_X.append(tf.identity(self.X))

#RNN
self.h_u = tf.zeros([M.shape[0], self.n_conv_feat])
self.c_u = tf.zeros([M.shape[0], self.n_conv_feat])
self.h_m = tf.zeros([M.shape[1], self.n_conv_feat])
self.c_m = tf.zeros([M.shape[1], self.n_conv_feat])
```

Define network parameters and initial matrix

Code explanation

Define multi-graph convolution network

Define LSTM network for **matrix diffusion**

Reconstruct matrix at each step of matrix diffusion

```
for k in range(self.num_iterations):
    #extraction of global features vectors
    self.final_feat_users = self.mono_conv(self.list_row_cheb_pol, self.ord_row,
                                           self.W, self.W_conv_W, self.b_conv_W)
    self.final_feat_movies = self.mono_conv(self.list_col_cheb_pol, self.ord_col,
                                            self.H, self.W_conv_H, self.b_conv_H)

    #here we have to split the features between users and movies LSTMs

    #users RNN
    self.f_u = tf.sigmoid(tf.matmul(self.final_feat_users, self.W_f_u)
                          + tf.matmul(self.h_u, self.U_f_u) + self.b_f_u)
    self.i_u = tf.sigmoid(tf.matmul(self.final_feat_users, self.W_i_u)
                          + tf.matmul(self.h_u, self.U_i_u) + self.b_i_u)
    self.o_u = tf.sigmoid(tf.matmul(self.final_feat_users, self.W_o_u)
                          + tf.matmul(self.h_u, self.U_o_u) + self.b_o_u)

    self.update_c_u = tf.sigmoid(tf.matmul(self.final_feat_users, self.W_c_u)
                                 + tf.matmul(self.h_u, self.U_c_u) + self.b_c_u)
    self.c_u = tf.multiply(self.f_u, self.c_u) + tf.multiply(self.i_u, self.update_c_u)
    self.h_u = tf.multiply(self.o_u, tf.sigmoid(self.c_u))

    #movies RNN
    self.f_m = tf.sigmoid(tf.matmul(self.final_feat_movies, self.W_f_m)
                          + tf.matmul(self.h_m, self.U_f_m) + self.b_f_m)
    self.i_m = tf.sigmoid(tf.matmul(self.final_feat_movies, self.W_i_m)
                          + tf.matmul(self.h_m, self.U_i_m) + self.b_i_m)
    self.o_m = tf.sigmoid(tf.matmul(self.final_feat_movies, self.W_o_m)
                          + tf.matmul(self.h_m, self.U_o_m) + self.b_o_m)

    self.update_c_m = tf.sigmoid(tf.matmul(self.final_feat_movies, self.W_c_m)
                                 + tf.matmul(self.h_m, self.U_c_m) + self.b_c_m)
    self.c_m = tf.multiply(self.f_m, self.c_m) + tf.multiply(self.i_m, self.update_c_m)
    self.h_m = tf.multiply(self.o_m, tf.sigmoid(self.c_m))

    #compute update of matrix X
    self.delta_W = tf.tanh(tf.matmul(self.c_u, self.W_out_W) + self.b_out_W) #N x rank_W_H
    self.delta_H = tf.tanh(tf.matmul(self.c_m, self.W_out_H) + self.b_out_H) #M x rank_W_H

    self.W += self.delta_W
    self.H += self.delta_H

    self.X = tf.matmul(self.W, self.H, transpose_b=True)
    self.list_X.append(tf.identity(tf.reshape(self.X, [tf.shape(self.M)[0], tf.shape(self.M)[1]])))
```

Code explanation

```
self.X = tf.matmul(self.W, self.H, transpose_b=True)
#########loss definition

#computation of the accuracy term
self.norm_X = 1+4*(self.X-tf.reduce_min(self.X))/(tf.reduce_max(self.X-tf.reduce_min(self.X)))
frob_tensor = tf.multiply(self.Otraining + self.Odata, self.norm_X - M)
self.loss_frob = tf.square(self.frobenius_norm(frob_tensor))/np.sum(Otraining+Odata)

#computation of the regularization terms
trace_col_tensor = tf.matmul(tf.matmul(self.X, self.Lc), self.X, transpose_b=True)
self.loss_trace_col = tf.trace(trace_col_tensor)
trace_row_tensor = tf.matmul(tf.matmul(self.X, self.Lr, transpose_a=True), self.X)
self.loss_trace_row = tf.trace(trace_row_tensor)

#training loss definition
self.loss = self.loss_frob + (gamma/2)*(self.loss_trace_col + self.loss_trace_row)

#test loss definition
self.predictions = tf.multiply(self.Otest, self.norm_X - self.M)
self.predictions_error = self.frobenius_norm(self.predictions)

#definition of the solver
self.optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(self.loss)

self.var_grad = tf.gradients(self.loss, tf.trainable_variables())
self.norm_grad = self.frobenius_norm(tf.concat([tf.reshape(g, [-1]) for g in self.var_grad], 0))
```

```
num_iter = 0
for k in range(num_iter, num_total_iter_training):

    tic = time.time()
    _, current_training_loss, norm_grad, X_grad = learning_obj.session.run(
        [learning_obj.optimizer, learning_obj.loss, learning_obj.norm_grad, learning_obj.var_grad]
    )
    training_time = time.time() - tic

    list_training_loss.append(current_training_loss)
    list_training_norm_grad.append(norm_grad)
    list_training_times.append(training_time)

    if (np.mod(num_iter, num_iter_test)==0):
        msg = "[TRN] iter = %03i, cost = %3.2e, |grad| = %.2e (%3.2es)" \
                                        % (num_iter, list_training_loss[-1], list_training_norm_grad[-1],
                                           training_time)
        print(msg)

        #Test Code
        tic = time.time()
        pred_error, preds, X = learning_obj.session.run(
            [learning_obj.predictions_error, learning_obj.predictions,learning_obj.norm_X]
        )
        c_X_evolutions = learning_obj.session.run(learning_obj.list_X)
        list_X_evolutions.append(c_X_evolutions)

        test_time = time.time() - tic

        list_test_pred_error.append(pred_error)
        list_X.append(X)
        list_test_times.append(test_time)

        RMSE = np.sqrt(np.square(pred_error)/np.sum(Otest))
        msg =  "[TST] iter = %03i, cost = %3.2e, RMSE = %3.2e (%3.2es)" \
                                        % (num_iter, list_test_pred_error[-1], RMSE, test_time)
        print(msg)

    num_iter += 1
```

Define the loss function and train the network parameters

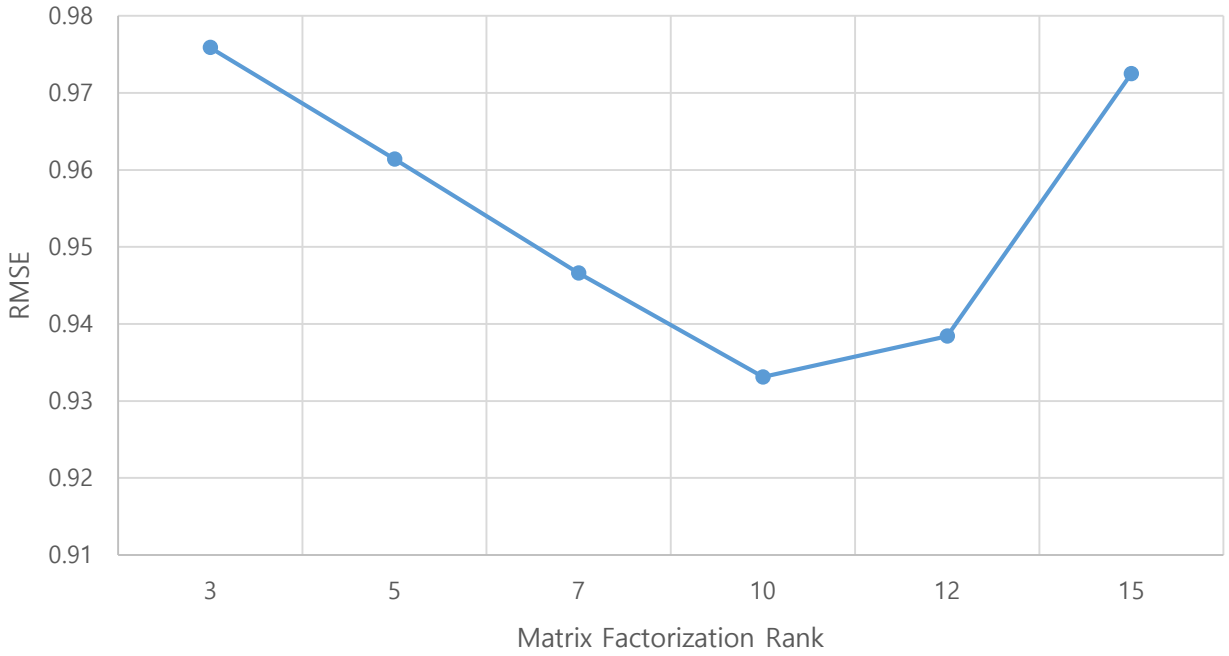
Reproduction results

Dataset	RMSE (paper)	RMSE (reproduction)
MovieLens	0.929	0.9331
Flixter	0.9258	0.9326
Douban	0.8012	0.7929
YahooMusic	22.4149	21.9876

For the all of datasets, the performance of the sRMGCNN was well reproduced.

Additional experiments

Performance of sRMGCNN changes as the rank of matrix factorization changes (Tested in MovieLens dataset).



Matrix Factorization Rank	RMSE
3	0.9759
5	0.9614
7	0.9466
10 (paper)	0.9331
12	0.9384
15	0.9725

The rank of 10 (which is used for experiments reported in the paper), shows the best performance.
The ranks larger than 10 may need to be trained for more training iterations.

Additional experiments

Performance of sRMGCNN using Chebyshev polynomial (order=5) and using Simplified ShevNet (Tested in MovieLens dataset).

Code :

```
def simplified_cheb_net(self, W, num):
    W_tilda = tf.eye(num) + W
    D_tilda = tf.linalg.diag(tf.sqrt(1.0 / tf.reduce_sum(W_tilda, axis=1)))
    return D_tilda * W_tilda * D_tilda, W_tilda, D_tilda
```

```
def sheb_conv(self, sheb_matrix, A, W, b):

    c_feat = tf.matmul(sheb_matrix, A)
    conv_feat = tf.matmul(c_feat, W) + b
    conv_feat = tf.nn.relu(conv_feat)

    return conv_feat
```

Graph Filter	RMSE	Training Time (s)
ChebNet (order=5)	0.9331	0.0363
Simplified ChebNet	1.019	0.0250

Using ShevNet shows faster training time, however it greatly degrades the performance (RMSE)

Conclusion

- Generalize a graph convolutional network into a **multi-graph convolutional network**
- Handle the matrix completion problem with **multi-graph CNN** + **matrix diffusion** using RNN structure
- **Simple & efficient** algorithm, seems to be practical to apply on various large-scale applications.

Thank You