Predicting Station-level Hourly Demand in a Large-scale bike-sharing Network: A Graph Convolutional Neural Network Approach

2019-24947
김지환

❖ PROBLEM

- Bike-sharing demand forecasting

kaggle

Search

Home
Compete
Data
Notebooks
Discuss
Courses
More

Recently Viewed
Bike Sharing Demand
M5 Forecasting - Accu...

Bike Sharing Demand

Forecast use of a city bikeshare system

3,251 teams · 5 years ago

Overview Data Notebooks Discussion Leaderboard Rules

Overview

Description
Evaluation

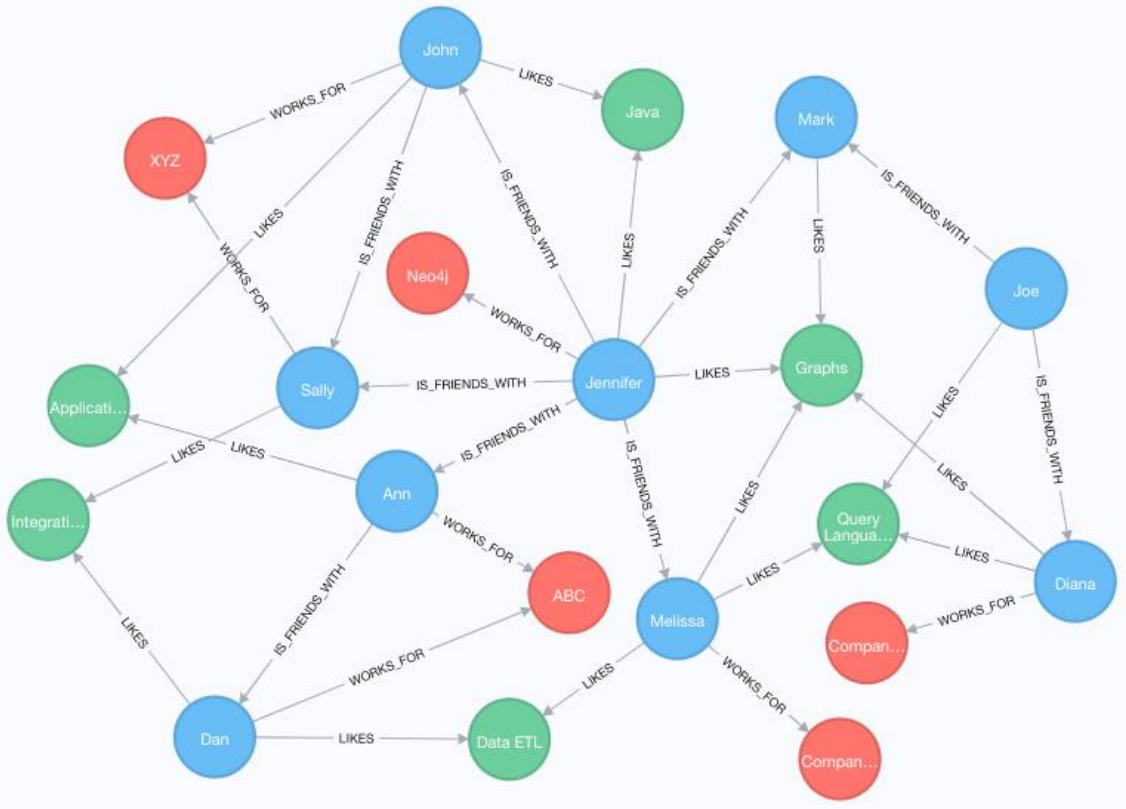
Get started on this competition through Kaggle Scripts

Bike sharing systems are a means of renting bicycles where the process and bike return is automated via a network of kiosk locations throughout people are able rent a bike from a one location and return it to a differen Currently, there are over 500 bike-sharing programs around the world.

서울자전거 SEOUL BIKE 따릉이

❖ PROBLEM

- Spatiotemporal data analysis via graph



Graph	Bike-sharing
Node	Station
Feature vector of Each node	Demand history of Each station
Edge	(depends on user. distance, correlation, etc)

❖ FRAMEWORK

- A graph convolutional neural network

3.1.1. Spectral convolution on graph

A spectral convolution on the graph is defined as follows:

$$g_\theta * x = U g_\theta(\Lambda) U^T x \tag{3}$$

where $g_\theta(\Lambda)$ is a function of the eigenvalues of L.

A form of polynomial filters has been used in a few studies (Defferrard et al., 2016; Kipf and Welling, 2016; Shuman et al., 2013):

$$g_\theta(\Lambda) = \sum_{k=0}^{K} \theta_k \Lambda^k \tag{4}$$

- In this paper, just use first-order polynomial

To improve the computational efficiency, Kipf and Welling (2016) simplified the calculation of $g_\theta * x$ by using only the first-order polynomial:

$$g_\theta * x \approx D^{-\frac{1}{2}} A D^{-\frac{1}{2}} x\theta \tag{6}$$

❖ FRAMEWORK

- A graph convolutional layer

For the output layer m, the result is:

$$H^m = D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H^{m-1} W^m \tag{10}$$

where $W^m \in \mathbb{R}^{C^{m-1} \times C^m}$ are the weight parameters to be learned, and $H^m \in \mathbb{R}^{N \times C^m}$ are the predictions, e.g., the bike-sharing demand of the N stations for the next hour when $C^m = 1$.

$$H^m = D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H^{m-1} W^m$$

We need to pre-define an adjacency matrix.
-> **BUT NOT TRIVIAL**

❖ FRAMEWORK

- Some possible adjacency matrix

3.2.1. Spatial distance matrix

$$A_{ij} = \begin{cases} 1 if \quad DIST_{ij} \le \kappa_{SD} \\ 0 if \quad DIST_{ij} > \kappa_{SD} \end{cases} \tag{11}$$

where $DIST_{ij}$ is the spatial distance between station i and j, and κ_{SD} is a pre-defined SD threshold.

3.2.2. Demand matrix

This approach makes use of the check in and check out station information in the bike-sharing transaction records. The symmetric demand matrix (DE), which considers the total demand between stations i and j, is built as follows:

$$DE_{ij} = \begin{cases} OD_{ij} + OD_{ji} if \quad i \ne j \\ OD_{ij} otherwise \end{cases} \tag{12}$$

where OD_{ij} is the aggregated demand from station i to station j.

For the bike-sharing graph network, if the total demand between two stations DE_{ij} is higher than a pre-defined threshold κ_{DE} the two stations are connected. In this way, a binary A can be built such that $A_{ij} = 1$ if $DE_{ij} \ge \kappa_{DE}$; otherwise $A_{ij} = 0$.

❖ FRAMEWORK

- Some possible adjacency matrix

3.2.3. Average trip duration matrix

This approach utilizes the trip duration information in the bike-sharing transaction records on the basis of the DE matrix. Each entry ATD_{ij} in an Average Trip Duration matrix (ATD) is defined as:

$$ATD_{ij} = TTD_{ij}/DE_{ij} \tag{13}$$

where TTD_{ij} is the total trip duration of all trips between stations i and j; DE_{ij} is an entry in the DE matrix.

Similarly, a binary adjacency matrix is formed such that $A_{ij} = 1$ if $ATD_{ij} \leq \kappa_{ATD}$, otherwise $A_{ij} = 0$. κ_{ATD} is a pre-defined threshold.

3.2.4. Demand correlation matrix

The Demand Correlation matrix (DC) is defined by calculating the Pearson Correlation Coefficient (PCC) based on the hourly bike demand series between station i and station j.

$$DC_{ij} = PCC(h_i, h_j) \tag{14}$$

where h_i and h_j are the hourly bike demand series for stations i and j.

In this approach, the bike-sharing graph network is built by connecting two stations with high PCC. Each entry of A is set to 1 if $DC_{ij} \geq \kappa_{DC}$, otherwise $A_{ij} = 0$; where κ_{DC} is the DC threshold.

❖ FRAMEWORK

- Correlations between stations are not trivial! -> Get it from data!

3.3.1. Data-driven graph filter

The predefinition of the adjacency matrix A is not trivial. The hidden correlations between stations may be heterogeneous. Hence, it may be hard to encode them using just one kind of metric such as the SD, DE, ATD or DC matrix. Now, suppose the adjacency matrix A is unknown; let $\widehat{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$, then (9) becomes:

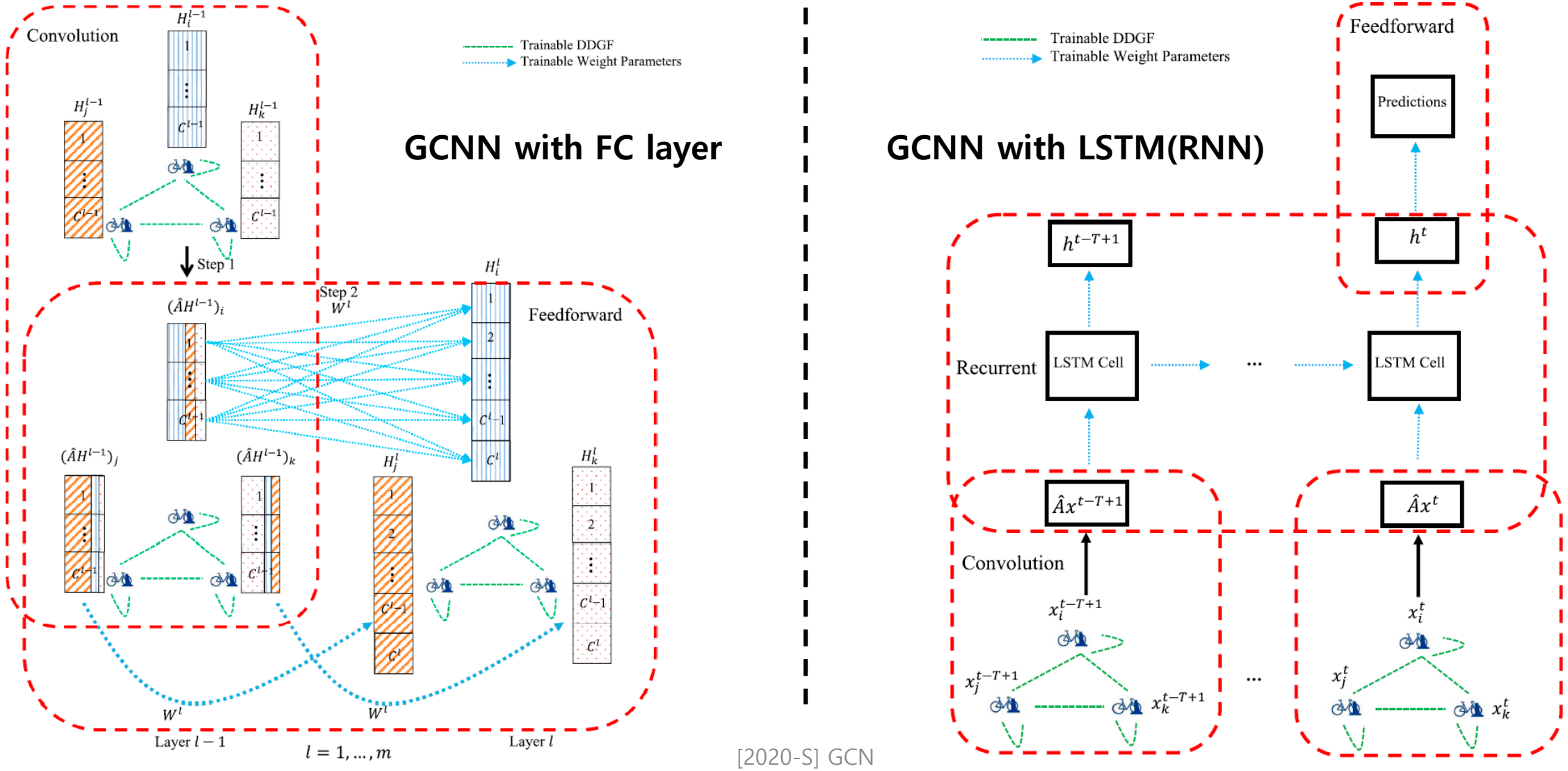
$$H^l = \sigma(\widehat{A} H^{l-1} W^l) \tag{15}$$

where $\widehat{A}$ is called the Data-driven Graph Filter (DDGF) which is a symmetric matrix consisting of trainable filter parameters, $\widehat{A} \in \mathbb{R}^{N \times N}$, $H^l \in \mathbb{R}^{N \times C^l}$.

Let DDGF $\hat{A}$ also be learnable parameter !

❖ FRAMEWORK

- Two deep learning architecture with GCNN-DDGF



❖ RESULT

- Forecasting accuracy

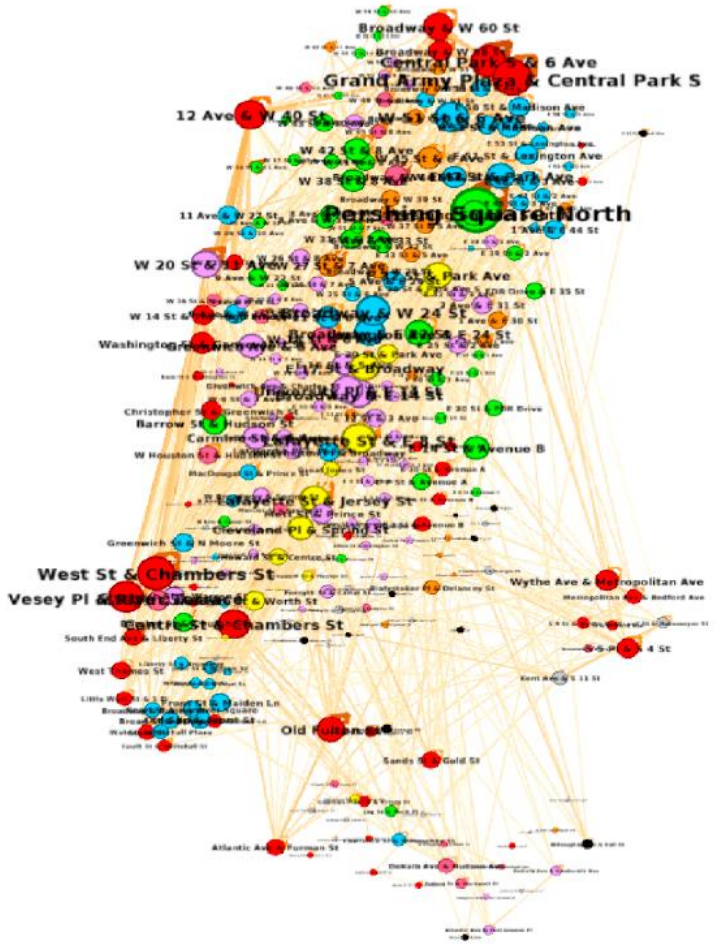
Table 3
Model comparison on testing dataset.

Model	RMSE	RMSE (7:00 AM–9:00 PM)	MAE	R^2
$GCNN_{rec}$-DDGF	2.12	2.58	1.26	0.75
$GCNN_{reg}$-DDGF	2.35	2.85	1.43	0.70
XGBoost	2.43	2.95	1.44	0.68
LSTM	2.46	3.00	1.44	0.67
GCNN-DC	2.50	3.02	1.53	0.66
MLP	2.51	3.05	1.51	0.65
GCNN-DE	2.67	3.21	1.60	0.61
SVR-RBF	2.67	3.25	1.57	0.61
LASSO	2.70	3.27	1.65	0.60
SVR-linear	2.72	3.31	1.52	0.59
GCNN-SD	2.77	3.31	1.68	0.58
HA	3.44	3.42	2.08	0.35
GCNN-ATD	3.44	3.83	2.21	0.35

Pre-defined A

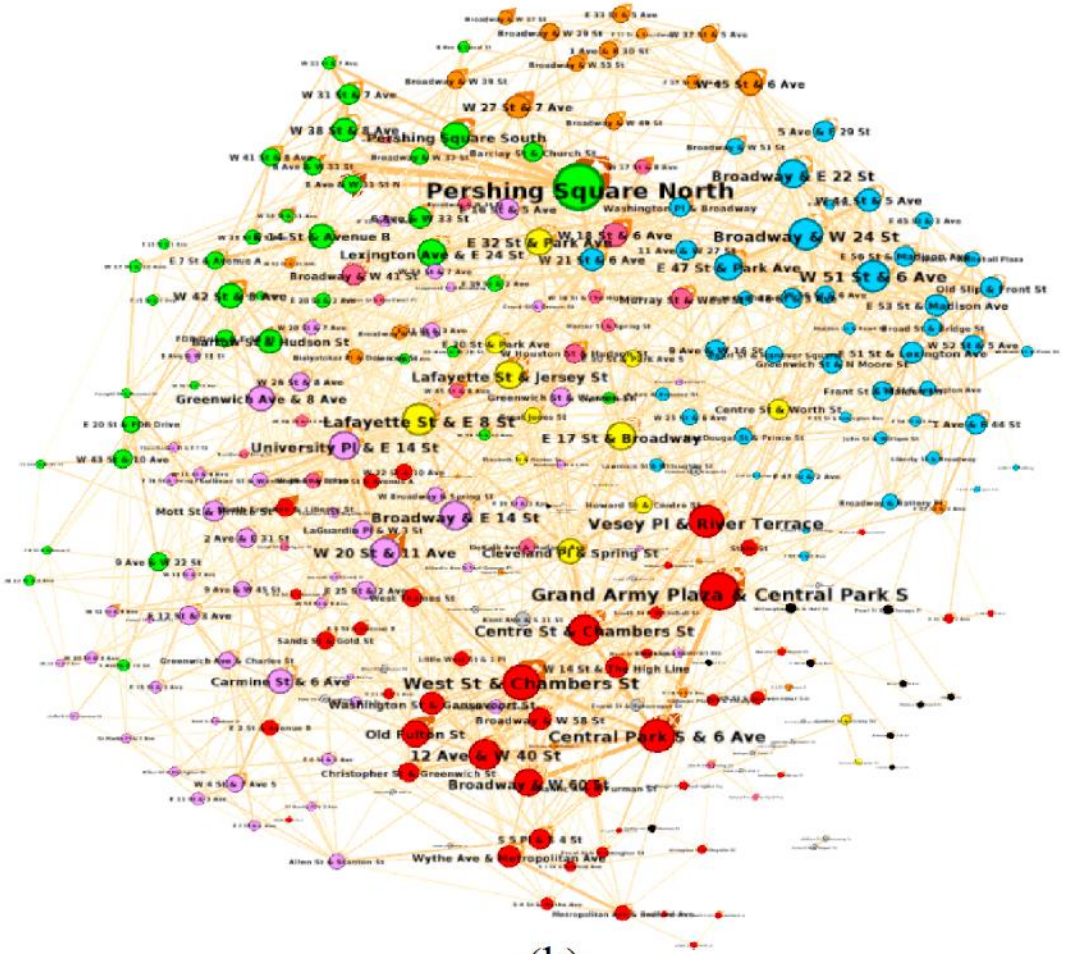
❖ RESULT

- Analyze correlations between station via $\hat{A}$



(a)

Geolocation Layout

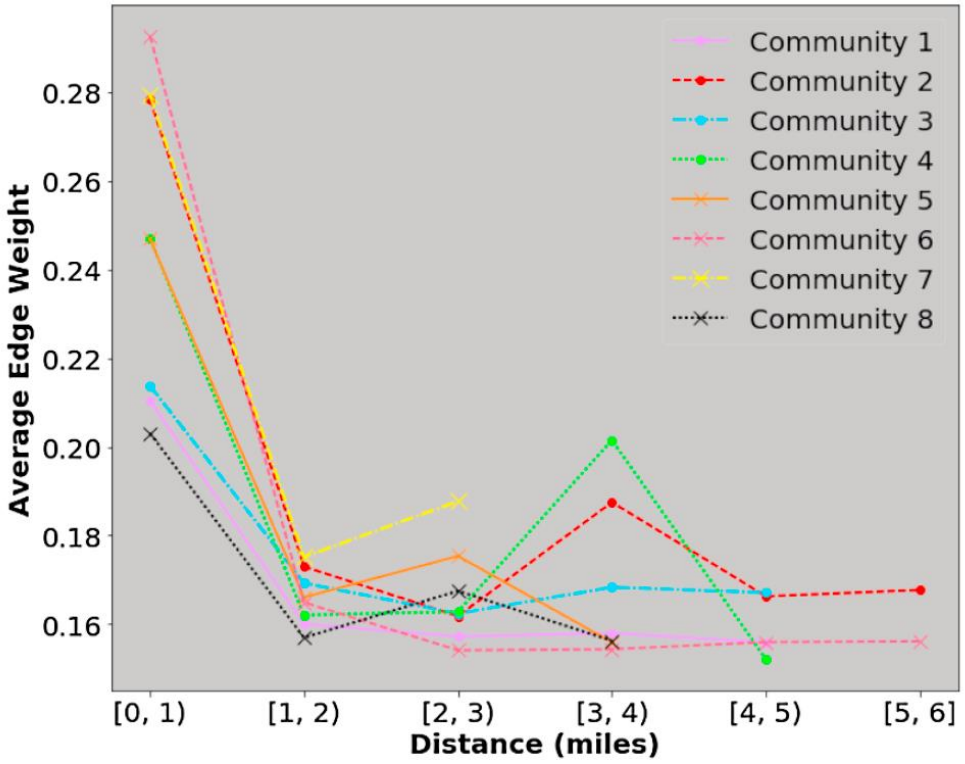


(b)

Force Altas Layout

❖ RESULT

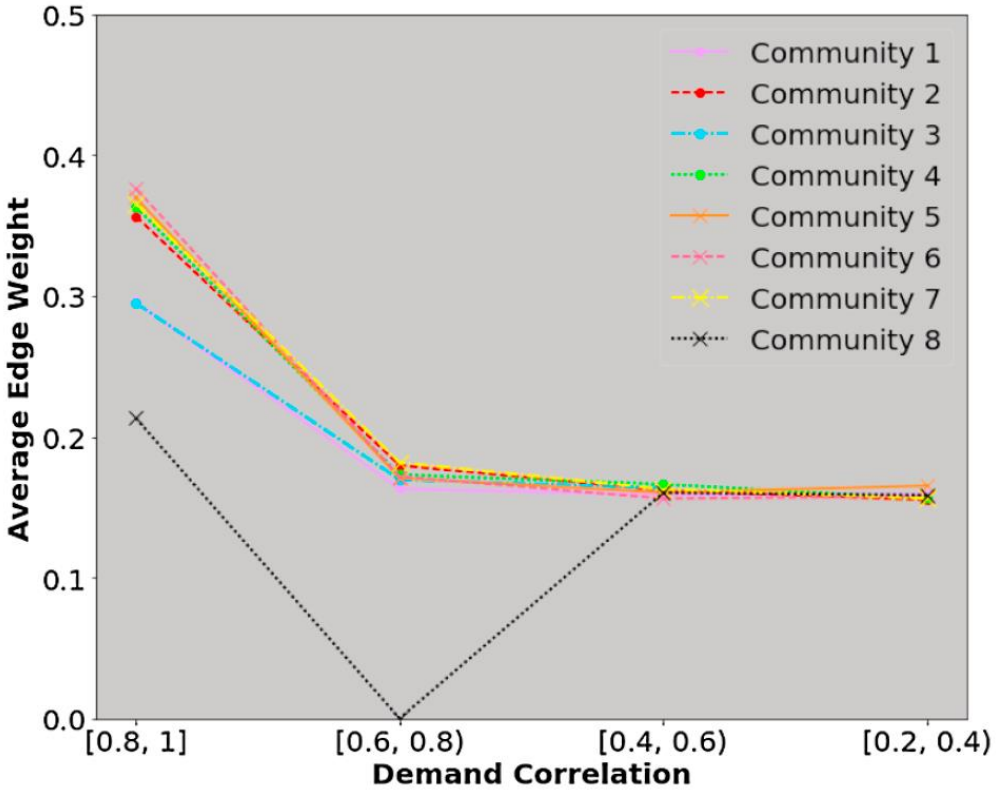
- DDGF vs Other affinity matrices



- The average edge weight is the largest when stations are spatially close to each other (0–1 miles)
- However, there also exist some fluctuations for a few communities.
- To some extent, the DDGF is like the SD ;
- However, the DDGF also reveals that the edge weight could still be large when two stations are far from each other.
- Therefore, the DDGF covers more heterogeneous pairwise information than the SD matrix.

❖ RESULT

- DDGF vs Other affinity matrices



- The average edge weight is the highest when the demand correlation coefficient is in the range of [0.8, 1] for all eight communities
- However, for other demand correlation ranges, the average edge weights are much lower, and the curves are almost flat.
- The correlations between stations based on the DDGF are consistent with the DE and the DC matrices to some extent.
- However, the nonlinear curves also indicate that the DDGF covers more heterogeneous pairwise information than these matrices.

❖ CONCLUSION

- A novel GCNN-DDGF model for station-level hourly demand prediction in a large-scale bike-sharing network

- Automatically capturing heterogeneous pairwise correlations between stations to improve prediction

- The DDGF not only captures some of the same information existing in the SD, DE and DC matrices, but also uncovers hidden correlations among stations that are not revealed by any of these matrices

Thank you for Listening

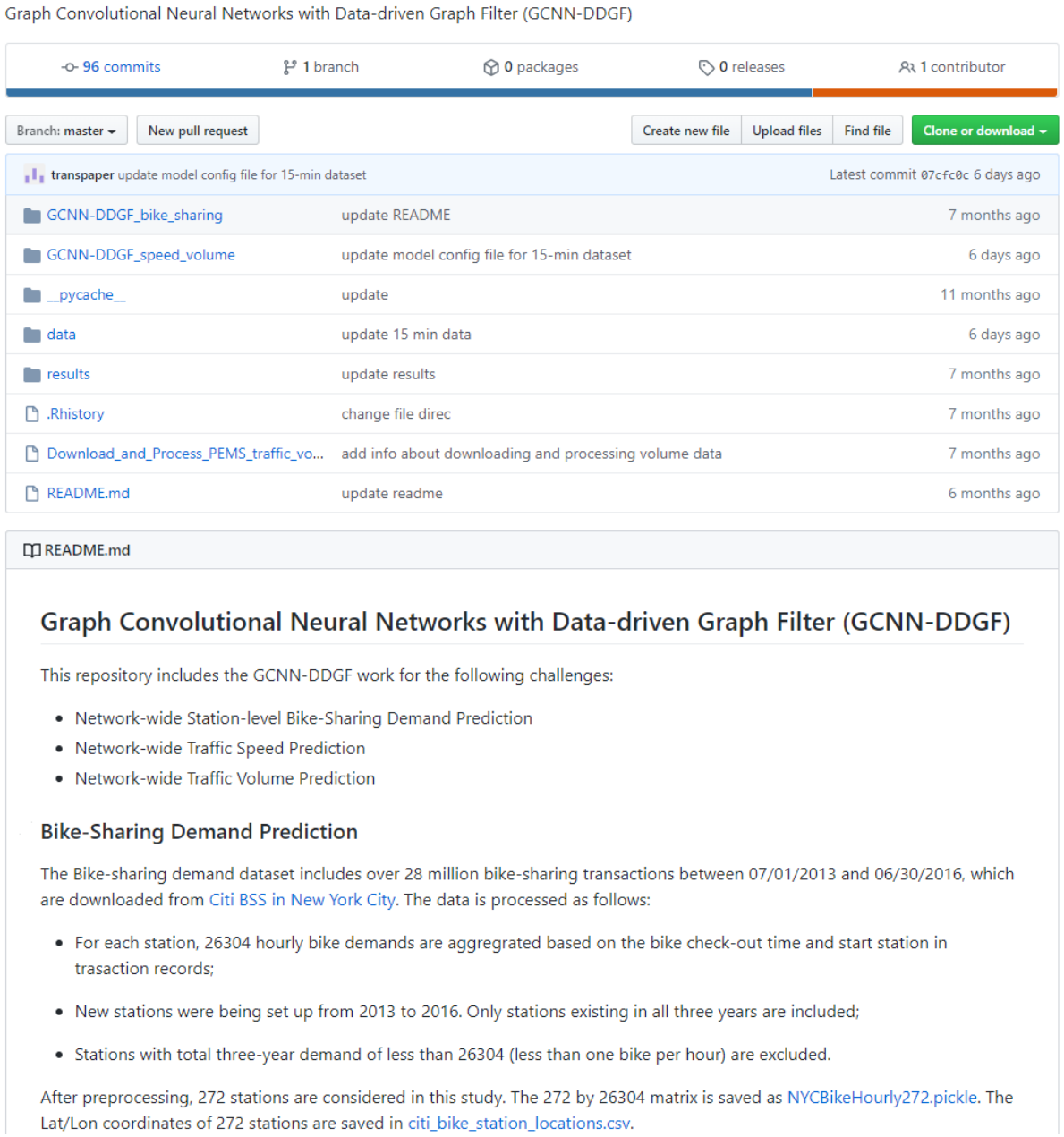
❖ Reproducing Results : Code

- Original Code from github of author.
- https://github.com/transpaper/GCGRNN

- Dependency in *requirements.txt*
- IMPORTANT : tensorflow=1.14.0

- Reproducing Code : github
- https://github.com/kjh6526/GCNfinal_GCNNDDGF

- Run *main.ipynb* in jupyter notebook / lab.



❖ Reproducing Results : main.ipynb

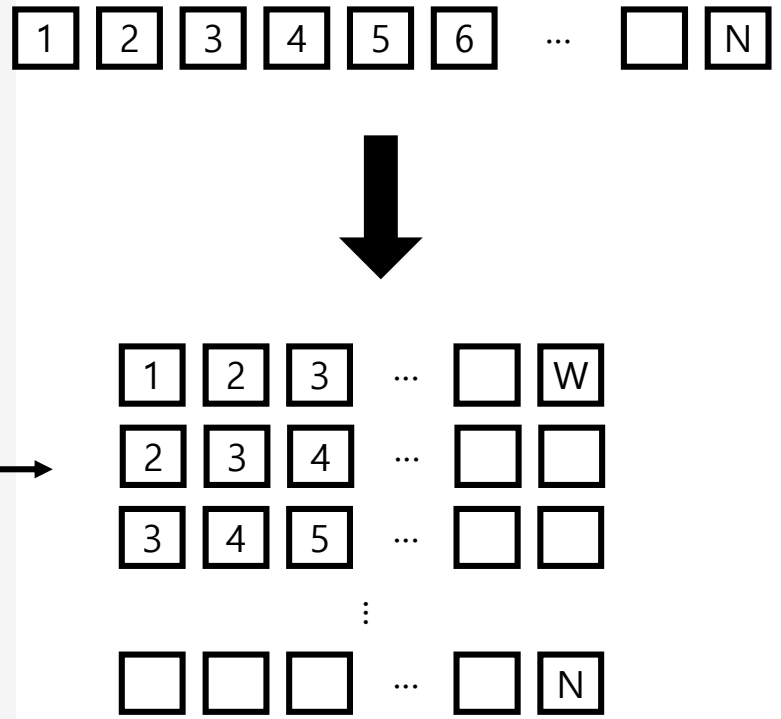
Import Data

[3]:
```
file_Name = "../data/NYC_Citi_bike/NYCBikeHourly272.pickle"
fileObject = open(file_Name,'rb')
hourly_bike = pickle.load(fileObject)
hourly_bike = pd.DataFrame(hourly_bike)
```

Split Data into Training, Validation and Testing ¶

[4]:
```
node_num = 272 # node number
feature_in = 24 # number of features at each node, e.g., bike sharing demand from past 24 hours
horizon = 1 # the length to predict, e.g., predict the future one hour bike sharing demand

X_whole = []
Y_whole = []

x_offsets = np.sort(
    np.concatenate((np.arange(-feature_in+1, 1, 1),))
)

y_offsets = np.sort(np.arange(1, 1+ horizon, 1))

min_t = abs(min(x_offsets))
max_t = abs(hourly_bike.shape[0] - abs(max(y_offsets)))  # Exclusive
for t in range(min_t, max_t):
    x_t = hourly_bike.iloc[t + x_offsets, 0:node_num].values.flatten('F')
    y_t = hourly_bike.iloc[t + y_offsets, 0:node_num].values.flatten('F')
    X_whole.append(x_t)
    Y_whole.append(y_t)

X_whole = np.stack(X_whole, axis=0)
Y_whole = np.stack(Y_whole, axis=0)

X_whole = np.reshape(X_whole, [X_whole.shape[0], node_num, feature_in])
```

Feature Making



❖ Reproducing Results : main.ipynb

[5] :

```
num_samples = X_whole.shape[0]
num_train = 20000
num_val = 2000
num_test = 2000

X_training = X_whole[:num_train, :]
Y_training = Y_whole[:num_train, :]

# shuffle the training dataset
perm = np.arange(X_training.shape[0])
np.random.shuffle(perm)
X_training = X_training[perm]
Y_training = Y_training[perm]

X_val = X_whole[num_train:num_train+num_val, :]
Y_val = Y_whole[num_train:num_train+num_val, :]

X_test = X_whole[num_train+num_val:num_train+num_val+num_test, :]
Y_test = Y_whole[num_train+num_val:num_train+num_val+num_test, :]


scaler = StandardScaler(mean=X_training.mean(), std=X_training.std())

X_training = scaler.transform(X_training)
Y_training = scaler.transform(Y_training)

X_val = scaler.transform(X_val)
Y_val = scaler.transform(Y_val)

X_test = scaler.transform(X_test)
Y_test = scaler.transform(Y_test)
```

Split Data for Train / Test / Validation

Data Normalization

❖ Reproducing Results : main.ipynb

Hyperparameters

[6]:
```
learning_rate = 0.01 # learning rate
decay = 0.9
batchsize = 100 # batch size

hidden_num_layer = [10, 10, 20] # determine the number of hidden layers and the vector length at each node of each hidden layer
reg_weight = [0, 0, 0] # regularization weights for adjacency matrices L1 loss

keep = 1 # drop out probability

early_stop_th = 200 # early stopping threshold, if validation RMSE not dropping in continuous 20 steps, break
training_epochs = 500 # total training epochs
```

$$H^l = \sigma\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}H^{l-1}W^l\right) \quad W^l \in \mathbb{R}^{C^{l-1}\times C^l}$$

of GCN layers $[C_1, C_2, \dots C_l, \dots C_M]$

Training

[7]:
```
start_time = datetime.datetime.now()

val_error, predic_res, test_Y, test_error, bestWeightA = gcnn_ddgf(hidden_num_layer, reg_weight, node_num, feature_in, horizon,
                                                                   learning_rate, decay, batchsize, keep, early_stop_th, training_epochs,
                                                                   X_training, Y_training, X_val, Y_val, X_test, Y_test, scaler, 'RMSE')

end_time = datetime.datetime.now()

print('Total training time: ', end_time-start_time)
```

❖ Reproducing Results : gcn.py

```python
# Create model
def gcn(signal_in, weights_hidden, weights_A, biases, hidden_num, node_num, horizon):

    signal_in = tf.transpose(signal_in, [1, 0, 2]) # node_num, ?batch, feature_in
    feature_len = signal_in.shape[2] # feature vector length at the node of the input graph

    i = 0
    while i < hidden_num:

        signal_in = tf.reshape(signal_in, [node_num, -1]) # node_num, batch*feature_in

        Adj = 0.5*(weights_A['A'+str(i)] + tf.transpose(weights_A['A'+str(i)]))
        Adj = normalize_adj(Adj)
        Z = tf.matmul(Adj, signal_in) # node_num, batch*feature_in
        Z = tf.reshape(Z, [-1, int(feature_len)]) # node_num * batch, feature_in
        signal_output = tf.add(tf.matmul(Z, weights_hidden['h'+str(i)]), biases['b'+str(i)])
        signal_output = tf.nn.relu(signal_output) # node_num * batch, hidden_vec

        i += 1
        signal_in = signal_output # the sinal for next layer
        feature_len = signal_in.shape[1] # feature vector length at hidden layers
        #print (feature_len)

    final_output = tf.add(tf.matmul(signal_output, weights_hidden['out']), biases['bout'])  # node_num * batch, horizon
    final_output = tf.reshape(final_output, [node_num, -1, horizon]) # node_num, batch, horizon
    final_output = tf.transpose(final_output, [1, 0, 2]) # batch, node_num, horizon
    final_output = tf.reshape(final_output, [-1, node_num*horizon]) # batch, node_num*horizon

    return final_output
```

Multiple GCN layers (in *while:*)

Symmetric Matrix (Affinity Matrix)

One Fully Connected Layer for output

❖ Reproducing Results : gcn.py

```
51 def gcnn_ddgf(hidden_num_layer, reg_weight, node_num, feature_in, horizon,
52               learning_rate, decay, batch_size, keep, early_stop_th, training_epochs,
53               X_training, Y_training, X_val, Y_val, X_test, Y_test, scaler, criterion):
54
55     n_output_vec = node_num * horizon # length of output vector at the final layer
56
57     early_stop_k = 0 # early stop patience
58     display_step = 1 # frequency of printing results
59     best_val = 10000
60     traing_error = 0
61     test_error = 0
62     predic_res = []
63     bestWeightA = {}
64
65     tf.reset_default_graph()
66
67     batch_size = batch_size
68     early_stop_th = early_stop_th
69     training_epochs = training_epochs
70
71     # tf Graph input and output
72     X = tf.placeholder(tf.float32, [None, node_num, feature_in]) # X is the input signal
73     Y = tf.placeholder(tf.float32, [None, n_output_vec]) # y is the regression output
74
75     keep_prob = tf.placeholder(tf.float32) #dropout (keep probability)
76
77     # define dictionaries to store layers weight & bias
78     i = 0
79     weights_hidden = {}
80     weights_A = {}
81     biases = {}
82     vec_length = feature_in
83     while i < len(hidden_num_layer):
84         weights_hidden['h'+str(i)] = tf.Variable(tf.random_normal([vec_length, hidden_num_layer[i]], stddev=0.5))
85         biases['b'+str(i)] = tf.Variable(tf.random_normal([1, hidden_num_layer[i]], stddev=0.5))
86         weights_A['A'+str(i)] = tf.Variable(tf.random_normal([node_num, node_num], stddev=0.5))
87         vec_length = hidden_num_layer[i]
88         i += 1
89
90
91     weights_hidden['out'] = tf.Variable(tf.random_normal([hidden_num_layer[-1], horizon], stddev=0.5))
92     biases['bout'] = tf.Variable(tf.random_normal([1, horizon], stddev=0.5))
```

Initialize all network weights.
(Learnable parameters)

Continued ...

❖ Reproducing Results : gcn.py

```
51 def gcnn_ddgf(hidden_num_layer, reg_weight, node_num, feature_in, horizon,
52              learning_rate, decay, batch_size, keep, early_stop_th, training_epochs,
53              X_training, Y_training, X_val, Y_val, X_test, Y_test, scaler, criterion):
```

⋮

```
94     # Construct model
95     hidden_num = len(hidden_num_layer)
96     pred = gcn(X, weights_hidden, weights_A, biases, hidden_num, node_num, horizon)
97     pred = scaler.inverse_transform(pred)
98     Y_original = scaler.inverse_transform(Y)
99
100    if criterion == 'RMSE':
101        cost = tf.sqrt(tf.reduce_mean(tf.pow(pred - Y_original, 2)))
102    elif criterion == 'MAE':
103        cost = masked_mae_tf(pred, Y_original, 0)
104    else:
105        print ('Please choose evaluation criterion from RMSE, MAE or MAPE!')
106        sys.exit()
107
108    # regularization
109    i = 0
110    while i < len(reg_weight):
111        cost += reg_weight[i]*tf.reduce_sum(tf.abs(weights_A['A'+str(i)]))
112        i += 1
113
114    #optimizer = tf.train.RMSPropOptimizer(learning_rate, decay).minimize(cost)
115    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
116
117    # Initializing the variables
118    init = tf.global_variables_initializer()
119    saver = tf.train.Saver()
```

Define the whole GCNN-DDGF network.

Set loss, regularizer, optimizer.

Continued ...

❖ Reproducing Results : gcn.py

```
51  def gcnn_ddgf(hidden_num_layer, reg_weight, node_num, feature_in, horizon,
52                learning_rate, decay, batch_size, keep, early_stop_th, training_epochs,
53                X_training, Y_training, X_val, Y_val, X_test, Y_test, scaler, criterion):
```

⋮

```
121     with tf.Session() as sess:
122         sess.run(init)
123
124         for epoch in range(training_epochs):
125
126             avg_cost = 0.
127             num_train = X_training.shape[0]
128             total_batch = int(num_train/batch_size)
129
130             for i in range(total_batch):
131
132                 _, c = sess.run([optimizer, cost], feed_dict={X: X_training[i*batch_size:(i+1)*batch_size,],
133                                                               Y: Y_training[i*batch_size:(i+1)*batch_size,],
134                                                                       keep_prob: keep})
135
136                 avg_cost += c * batch_size #/ total_batch
137
138             # rest part of training dataset
139             if total_batch * batch_size != num_train:
140                 _, c = sess.run([optimizer, cost], feed_dict={X: X_training[total_batch*batch_size:num_train,],
141                                                               Y: Y_training[total_batch*batch_size:num_train,],
142                                                                       keep_prob: keep})
143                 avg_cost += c * (num_train - total_batch*batch_size)
144
145             avg_cost = avg_cost / num_train
146
147             #Display logs per epoch step
148             if epoch % display_step == 0:
149                 print ("Epoch:", '%04d' % (epoch+1), "Training " + criterion+ " =",                    "{:.9f}".format(avg_cost))
150             # validation
151             c_val = sess.run([cost], feed_dict={X: X_val, Y: Y_val,  keep_prob:1})
152             c_val = c_val[0]
153             print("Validation " + criterion+":", c_val)
154             # testing
155             c_tes, preds, Y_true, weights_A_final = sess.run([cost, pred, Y_original, weights_A], feed_dict={X: X_test,Y: Y_test, keep_prob: 1})
```

Batch Training

Calculate Val, Test loss

Continued ...

❖ Reproducing Results : gcn.py

```
51  def gcnn_ddgf(hidden_num_layer, reg_weight, node_num, feature_in, horizon,
52                learning_rate, decay, batch_size, keep, early_stop_th, training_epochs,
53                X_training, Y_training, X_val, Y_val, X_test, Y_test, scaler, criterion):
                                        ⋮
158             if c_val < best_val:
159                 best_val = c_val
160                 # save model
161                 #saver.save(sess, './bikesharing_gcnn_ddgf')
162                 test_error = c_tes
163                 traing_error = avg_cost
164                 predic_res = preds
165                 bestWeightA = weights_A_final
166                 early_stop_k = 0 # reset to 0
167
168             # update early stopping patience
169             if c_val >= best_val:
170                 early_stop_k += 1
171
172             # threshold
173             if early_stop_k == early_stop_th:
174                 break
175
176
177         print("epoch is ", epoch)
178         print("training " + criterion +" is ", traing_error)
179         print("Optimization Finished! the lowest validation " + criterion + " is ", best_val)
180         print("The test " + criterion + " is ", test_error)
181
182     #test_Y = Y_test
183     #test_error = np.sqrt(test_error)
184     return best_val, predic_res,Y_true,test_error, bestWeightA
185
```

Stopping criteria

❖ Reproducing Results : Run main.ipynb

Training ¶

```
[7]: start_time = datetime.datetime.now()

val_error, predic_res, test_Y, test_error, bestWeightA = gcnn_ddgf(hidden_num_layer, reg_weight, node_num, feature_in, horizon,
                                                                   learning_rate, decay, batchsize, keep, early_stop_th, training_epochs,
                                                                   X_training, Y_training, X_val, Y_val, X_test, Y_test, scaler, 'RMSE')
end_time = datetime.datetime.now()

print('Total training time: ', end_time-start_time)
```

```
Epoch: 0001 Training RMSE = 3.936679145
Validation RMSE: 3.6874359
Epoch: 0002 Training RMSE = 3.422823601
Validation RMSE: 3.623373
Epoch: 0003 Training RMSE = 3.195017954
Validation RMSE: 3.508453
Epoch: 0004 Training RMSE = 3.100379407
Validation RMSE: 3.5189042
Epoch: 0005 Training RMSE = 3.056049792
Validation RMSE: 3.4705627
                  ⋮
Epoch: 0498 Training RMSE = 2.585240983
Validation RMSE: 2.985249
Epoch: 0499 Training RMSE = 2.584903015
Validation RMSE: 2.984735
Epoch: 0500 Training RMSE = 2.584538321
Validation RMSE: 2.9847744
epoch is  499
training RMSE is  2.597171092033386
Optimization Finished! the lowest validation RMSE is  2.9713192
The test RMSE is  2.3411646
Total training time:  0:05:43.988986
```

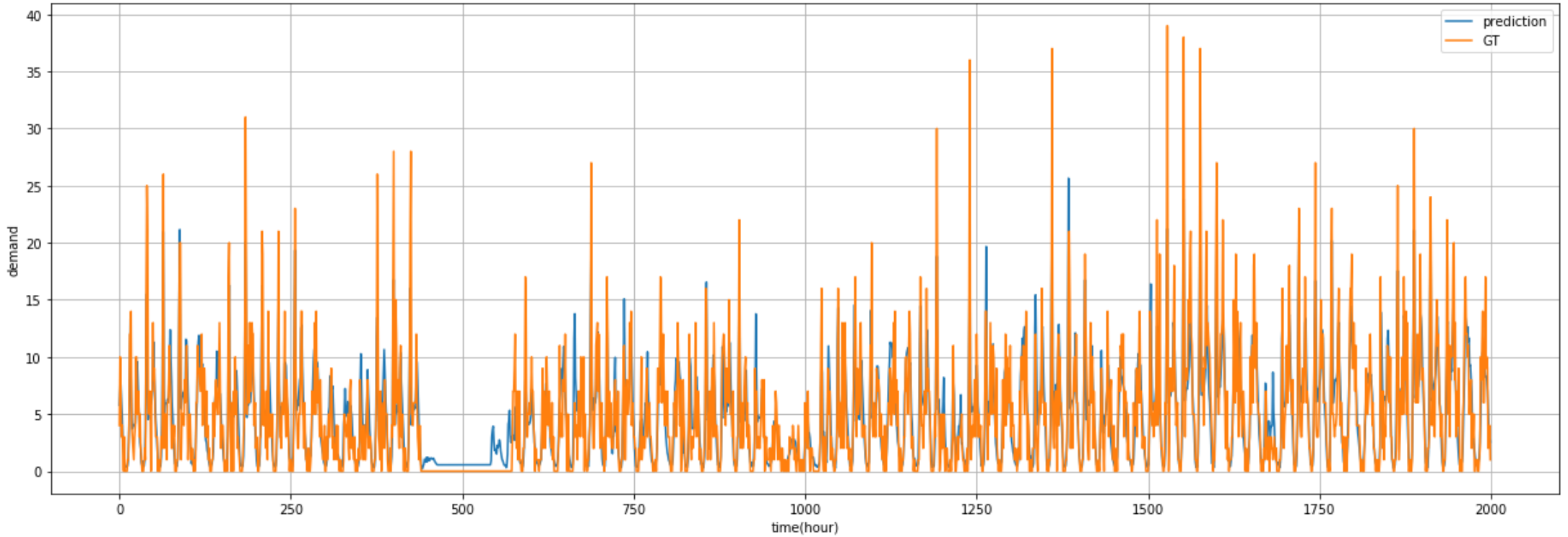
Table 3
Model comparison on testing dataset.

Model	RMSE	RMSE (7:00 AM–9:00 PM)	MAE	R^2
$GCNN_{rec}$-DDGF	2.12	2.58	1.26	0.75
$GCNN_{reg}$-DDGF	2.35	2.85	1.43	0.70
XGBoost	2.43	2.95	1.44	0.68
LSTM	2.46	3.00	1.44	0.67
GCNN-DC	2.50	3.02	1.53	0.66
MLP	2.51	3.05	1.51	0.65
GCNN-DE	2.67	3.21	1.60	0.61
SVR-RBF	2.67	3.25	1.57	0.61
LASSO	2.70	3.27	1.65	0.60
SVR-linear	2.72	3.31	1.52	0.59
GCNN-SD	2.77	3.31	1.68	0.58
HA	3.44	3.42	2.08	0.35
GCNN-ATD	3.44	3.83	2.21	0.35

❖ Reproducing Results : Run main.ipynb

[8]:
```
idx = 0 # Node(Station) index
plt.figure(figsize=(21, 7))
plt.plot(predic_res[:, idx], label='prediction')
plt.plot(test_Y[:, idx], label='GT')

# plt.xlim(24*5, 24*15)
# plt.ylim(-2,35)

plt.xlabel('time(hour)')
plt.ylabel('demand')
plt.legend()
plt.grid()
plt.show()
```

Prediction Result Plot



❖ Reproducing Results : Plot the prediction result