Performance Comparison of Univariate CNN and LSTM Neural Network for Predicting Particulate Matter (PM2.5) Concentration

> By: 05111950010040 - Hafara Firdausi

1. Description

1.1 Technologies Used

- Python 3.7.3
- Python Libraries :
  - Pandas
  - Numpy
  - Matplotlib
  - Seaborn
  - TensorFlow
  - Scikit-learn
- Others:
  - Visual Studio Code
  - Jupyter Notebook

1.2 Dataset

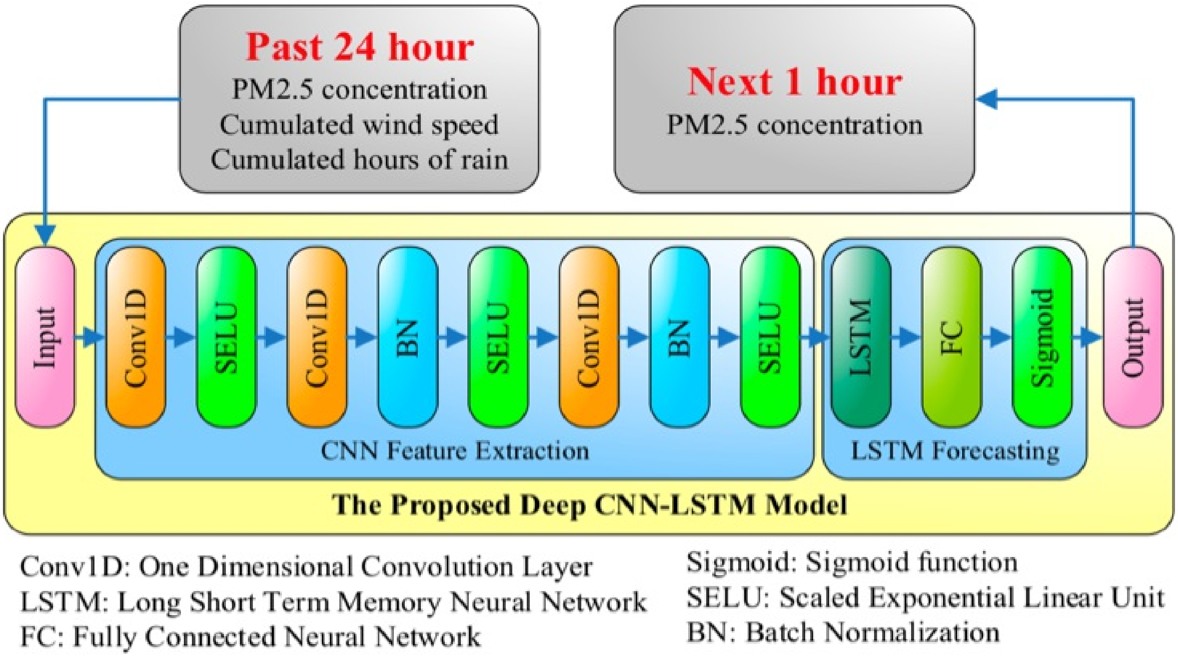
The dataset used in this final project is a real dataset from UCI Machine Learning titled "Beijing PM2.5 Data Data Set". It contains the PM2.5 data of the US Embassy in Beijing collected between Jan 1st, 2010 to Dec 31st, 2014 (4 years). It has 43824 rows of data and 13 attributes with details as follows:

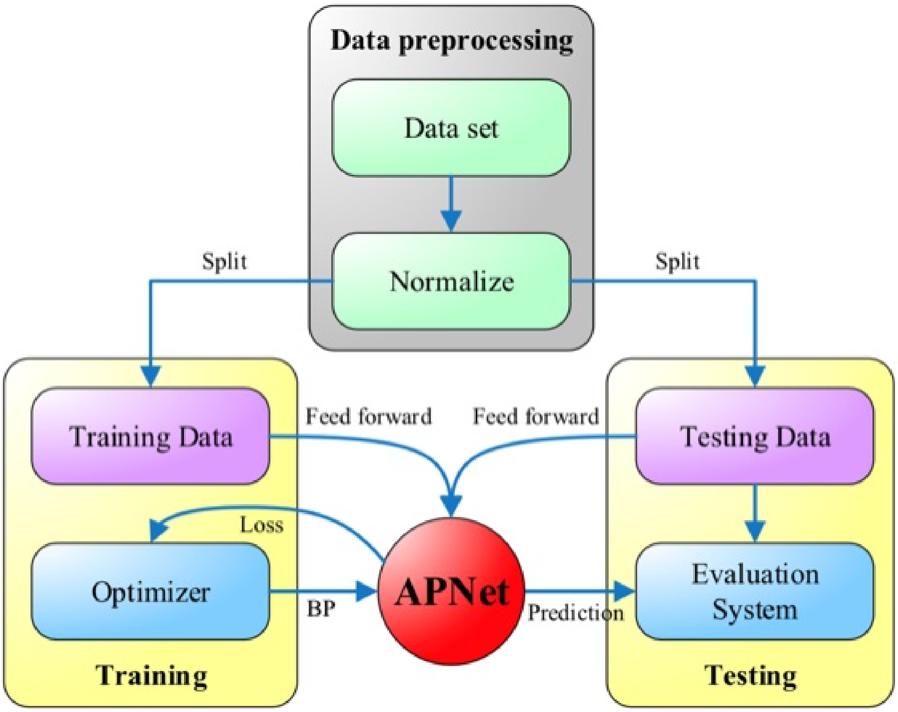
No.	Attribute	Description
1	No	Row number
2	Year	Year of data in this row
3	Month	Month of data in this row
4	Day	Day of data in this row
5	Hour	Hour of data in this row
6	pm2.5	PM2.5 concentration (ug/m^3)
7	DEWP	Dew Point (â„ƒ)
8	TEMP	Temperature (â„ƒ)
9	PRES	Pressure (hPa)
10	cbwd	Combined wind direction
11	Iws	Cumulated wind speed (m/s)
12	Is	Cumulated hours of snow
13	Ir	Cumulated hours of rain

This final project only uses 1 attribute (univariate) which is **pm2.5 concentration** to predict the next pm2.5 concentration.

2. Methodology

Initially, the proposed method for predicting PM2.5 concentrations in the air was a CNN-LSTM model proposed by C.-J. Huang and P.-H. Kuo in 2018. It is a model combining the CNN model for feature extraction and the LSTM model for prediction. The detailed architecture of the CNN-LSTM model shown in Figure 1. Also, the flowchart of the prediction system shown in Figure 2. However, CNN-LSTM architecture cannot be implemented yet in this final project.





Hence, this final project implements both the univariate CNN model and the univariate LSTM model individually to predict PM2.5 concentration in the air. Then, the performance of the two models compared using MAE (Mean Absolute Error) and RMSE (Root Mean Square Error) metrics.

3. Implementation

3.1 Preprocessing

First, the raw data must be preprocessed before used for forecasting. In this case, I only fill in the missing value with zero.

In [1]:

```
# import all libraries needed
import os, sys, math
from datetime import datetime
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, Conv1D, MaxPooling1D, Flatten
from tensorflow.keras.callbacks import EarlyStopping
```

In [2]:

```
"""
Data Handling
"""
# function to load data
def load_data(path):
    # read all column names
    cols = list(pd.read_csv(path, nrows =1))
    df = pd.read_csv(path, sep=',', header=0, low_memory=False, infer_datetime_format=True, parse_d
ates={'datetime':['year', 'month', 'day', 'hour']}, date_parser=custom_parser, index_col=['datetime
'], usecols=[i for i in cols if i != 'No'])
    # specify new column names
    df.columns = ['pollution', 'dew', 'temp', 'press', 'wnd_dir', 'wnd_spd', 'snow', 'rain']
    return df

# function to export data
def export_data(df, path):
  df.to_csv(path)
  print('>>> data exported succesfully!')

# function to get metadata
def get_metadata(df):
  return {
    'row_num' : df.shape[0],
    'col_num' : df.shape[1],
    'attr' : df.columns.to_list(),
    'timeseries_start': df.index.min(),
    'timeseries_end': df.index.max(),
    'null_value': df.isnull().sum().to_dict(),
    'dtypes': df.dtypes.to_dict()
  }

# function to parse datetime, specifically for this case
def custom_parser(year, month, day, hour):
    date_string = year + ' ' + month + ' ' + day + ' ' + hour
    return datetime.strptime(date_string, '%Y %m %d %H')

# function to print metadata
def print_metadata(metadata):
    print('=====================================')
    print('[ METADATA ]')
    for key, val in metadata.items():
        print('{} => {}'.format(key, val))
    print('=====================================')
```

In [3]:

```
# define path
PATH = os.getcwd()
DATASET_DIR_PATH = PATH + '/../data/'
DATASET_PATH = {
    'raw': DATASET_DIR_PATH + 'PRSA_data.csv',
    'preprocessed': DATASET_DIR_PATH + 'PRSA_data_preprocessed.csv'
}
```

In [4]:

```
# get initial data and metadata
df = load_data(DATASET_PATH['raw'])
metadata = get_metadata(df)

df.head(5)
print_metadata(metadata)
```

```
=====================================
[ METADATA ]
row_num => 43824
col_num => 8
attr => ['pollution', 'dew', 'temp', 'press', 'wnd_dir', 'wnd_spd', 'snow', 'rain']
```

```
timeseries_start => 2010-01-01 00:00:00
timeseries_end => 2014-12-31 23:00:00
null_value => {'pollution': 2067, 'dew': 0, 'temp': 0, 'press': 0, 'wnd_dir': 0, 'wnd_spd': 0,
'snow': 0, 'rain': 0}
dtypes => {'pollution': dtype('float64'), 'dew': dtype('int64'), 'temp': dtype('float64'),
'press': dtype('float64'), 'wnd_dir': dtype('O'), 'wnd_spd': dtype('float64'), 'snow':
dtype('int64'), 'rain': dtype('int64')}
=====================================
```

In [5]:

```
# get attr that has null value based on the metadata
nullAttr = []
for key, val in metadata['null_value'].items():
    if val > 0:
        nullAttr.append(key)

# fill missing values
for attr in nullAttr:
    df[attr].fillna(0, inplace=True)

# get new metadata
metadata = get_metadata(df)
print_metadata(metadata)
```

```
=====================================
[ METADATA ]
row_num => 43824
col_num => 8
attr => ['pollution', 'dew', 'temp', 'press', 'wnd_dir', 'wnd_spd', 'snow', 'rain']
timeseries_start => 2010-01-01 00:00:00
timeseries_end => 2014-12-31 23:00:00
null_value => {'pollution': 0, 'dew': 0, 'temp': 0, 'press': 0, 'wnd_dir': 0, 'wnd_spd': 0,
'snow': 0, 'rain': 0}
dtypes => {'pollution': dtype('float64'), 'dew': dtype('int64'), 'temp': dtype('float64'),
'press': dtype('float64'), 'wnd_dir': dtype('O'), 'wnd_spd': dtype('float64'), 'snow':
dtype('int64'), 'rain': dtype('int64')}
=====================================
```

In [6]:

```
# drop the first 24 hours or 1 day because the pollution table is 0
df = df[24:]
# get new metadata
metadata = get_metadata(df)
print_metadata(metadata)
```

```
=====================================
[ METADATA ]
row_num => 43800
col_num => 8
attr => ['pollution', 'dew', 'temp', 'press', 'wnd_dir', 'wnd_spd', 'snow', 'rain']
timeseries_start => 2010-01-02 00:00:00
timeseries_end => 2014-12-31 23:00:00
null_value => {'pollution': 0, 'dew': 0, 'temp': 0, 'press': 0, 'wnd_dir': 0, 'wnd_spd': 0,
'snow': 0, 'rain': 0}
dtypes => {'pollution': dtype('float64'), 'dew': dtype('int64'), 'temp': dtype('float64'),
'press': dtype('float64'), 'wnd_dir': dtype('O'), 'wnd_spd': dtype('float64'), 'snow':
dtype('int64'), 'rain': dtype('int64')}
=====================================
```

In [7]:

```
# export data
export_data(df, DATASET_PATH['preprocessed'])
```

```
>>> data exported succesfully!
```

3.2 Forecasting

The steps are:

- Preparing
- Splitting
- Training
- Evaluation

3.2.1 Preparing

- Convert pandas data frame into an array
- Reshape to adjust the dimension of data
- Normalize data with range 0 to 1

In [61]:

```
# import the preprocessed data
df_process = pd.read_csv(DATASET_PATH['preprocessed'], header=0, index_col=0)
```

In [62]:

```
df_process.head()
```

Out[62]:

	pollution	dew	temp	press	wnd_dir	wnd_spd	snow	rain
datetime								
2010-01-02 00:00:00	129.0	-16	-4.0	1020.0	SE	1.79	0	0
2010-01-02 01:00:00	148.0	-15	-4.0	1020.0	SE	2.68	0	0
2010-01-02 02:00:00	159.0	-11	-5.0	1021.0	SE	3.57	0	0
2010-01-02 03:00:00	181.0	-7	-5.0	1022.0	SE	5.36	1	0
2010-01-02 04:00:00	138.0	-7	-5.0	1022.0	SE	6.25	2	0

In [63]:

```
# save data values as array
val = df_process['pollution'].values
print(val)
print('dimension = {}'.format(val.shape))
```

```
[129. 148. 159. ...  10.   8.  12.]
dimension = (43800,)
```

In [64]:

```
# reshape to change the dimension of data
val = np.reshape(val, (-1, 1))
print('dimension = {}'.format(val.shape))
```

```
dimension = (43800, 1)
```

In [65]:

```
# normalize data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_val = scaler.fit_transform(val)
print(scaled_val)
```

```
[[0.12977867]
 [0.14889336]
 [0.15995976]
 ...
 [0.01006036]
 [0.00804829]
 [0.01207243]]
```

3.2.2 Splitting

- Split data into train and test data
- Split data into input and output data

In [66]:

```
# determine train and test size
# train size = 20%
# test size = 80%

train_size = int(len(val) * 0.20)
test_size = len(val) - train_size
print('Train size: ', train_size)
print('Test size: ', test_size)
```

```
Train size:  8760
Test size:  35040
```

In [67]:

```
# split data into train and test
data = {
    'train': scaled_val[0:train_size,:],
    'test': scaled_val[train_size:len(scaled_val),:]
}

for key, val in data.items():
    print('{} => {}'.format(key, val))
```

```
train => [[0.12977867]
 [0.14889336]
 [0.15995976]
 ...
 [0.        ]
 [0.        ]
 [0.        ]]
test => [[0.0362173 ]
 [0.03118712]
 [0.02012072]
 ...
 [0.01006036]
 [0.00804829]
 [0.01207243]]
```

In [68]:

```
# function to split data into input and output
def split_sequence(sequence, n_steps):
    X, y = [], []
    for i in range(len(sequence)):
        # find the end of this pattern
        end_ix = i + n_steps
        # check if we are beyond the sequence
        if end_ix > len(sequence)-1:
            break
        # gather input and output parts of the pattern
        seq_x, seq_y = sequence[i:end_ix, 0], sequence[end_ix, 0]
        X.append(seq_x)
        y.append(seq_y)
    return np.array(X), np.array(y)
```

In [69]:

```
# split data into input and output
# using the previous 24 hours as input
n_steps = 24

data['train_in'], data['train_out'] = split_sequence(data['train'], n_steps)
```

```
data['test_in'], data['test_out'] = split_sequence(data['test'], n_steps)
```

In [70]:

```
for key, val in data.items():
    print('{} => {}'.format(key, val.shape))
```

```
train => (8760, 1)
test => (35040, 1)
train_in => (8736, 24)
train_out => (8736,)
test_in => (35016, 24)
test_out => (35016,)
```

In [71]:

```
# data train input
pd.DataFrame(data['train_in']).head()
```

Out[71]:

	0	1	2	3	4	5	6	7	8	9	...	14	15	1
0	0.129779	0.148893	0.159960	0.182093	0.138833	0.109658	0.105634	0.124748	0.120724	0.132797	...	0.158954	0.154930	0.15996
1	0.148893	0.159960	0.182093	0.138833	0.109658	0.105634	0.124748	0.120724	0.132797	0.140845	...	0.154930	0.159960	0.16499
2	0.159960	0.182093	0.138833	0.109658	0.105634	0.124748	0.120724	0.132797	0.140845	0.152918	...	0.159960	0.164990	0.17102
3	0.182093	0.138833	0.109658	0.105634	0.124748	0.120724	0.132797	0.140845	0.152918	0.148893	...	0.164990	0.171026	0.14989
4	0.138833	0.109658	0.105634	0.124748	0.120724	0.132797	0.140845	0.152918	0.148893	0.164990	...	0.171026	0.149899	0.15493

5 rows × 24 columns

In [72]:

```
# data train output
pd.DataFrame(data['train_out']).head()
```

Out[72]:

	0
0	0.090543
1	0.063380
2	0.065392
3	0.055332
4	0.065392

3.2.3 LSTM Model

- 100 neurons LSTM in the first layer
- 1 neuron in the output layer to predict PM2.5 concentration
- Model fitted with training epochs = 50 and batch size = 70

In [53]:

```
# reshape input menjadi 3D array [samples, time steps, features]
data['train_in'] = np.reshape(data['train_in'], (data['train_in'].shape[0], 1, data['train_in'] .sh
ape[1]))
data['test_in']  = np.reshape(data['test_in'], (data['test_in'].shape[0], 1, data['test_in'].shape[
1]))
```

In [54]:

```
# design model
```

```
model = Sequential()
model.add(LSTM(100, input_shape=(data['train_in'].shape[1], data['train_in'].shape[2])))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
```

3.2.3.1 Training

In [55]:

```
# train data and recorded in history
history = model.fit(data['train_in'], data['train_out'], epochs=50, batch_size=70,
                    validation_data=(data['test_in'], data['test_out']), verbose=1, shuffle=False)
```

```
Train on 8736 samples, validate on 35016 samples
Epoch 1/50
8736/8736 [==============================] - 2s 235us/sample - loss: 0.0031 - val_loss: 0.0016
Epoch 2/50
8736/8736 [==============================] - 1s 125us/sample - loss: 0.0015 - val_loss: 0.0014
Epoch 3/50
8736/8736 [==============================] - 1s 127us/sample - loss: 0.0013 - val_loss: 0.0010
Epoch 4/50
8736/8736 [==============================] - 1s 123us/sample - loss: 0.0011 - val_loss: 9.4051e-04
Epoch 5/50
8736/8736 [==============================] - 1s 124us/sample - loss: 0.0011 - val_loss: 0.0011
Epoch 6/50
8736/8736 [==============================] - 1s 124us/sample - loss: 0.0011 - val_loss: 0.0011
Epoch 7/50
8736/8736 [==============================] - 1s 125us/sample - loss: 0.0011 - val_loss: 9.4767e-04
Epoch 8/50
8736/8736 [==============================] - 1s 125us/sample - loss: 0.0011 - val_loss: 9.8689e-04
Epoch 9/50
8736/8736 [==============================] - 1s 125us/sample - loss: 0.0011 - val_loss: 9.2583e-04
Epoch 10/50
8736/8736 [==============================] - 1s 129us/sample - loss: 0.0010 - val_loss: 9.3520e-04
Epoch 11/50
8736/8736 [==============================] - 1s 128us/sample - loss: 0.0010 - val_loss: 9.1478e-04
Epoch 12/50
8736/8736 [==============================] - 1s 128us/sample - loss: 0.0010 - val_loss: 9.1158e-04
Epoch 13/50
8736/8736 [==============================] - 1s 128us/sample - loss: 0.0010 - val_loss: 9.0365e-04
Epoch 14/50
8736/8736 [==============================] - 1s 127us/sample - loss: 0.0010 - val_loss: 8.9910e-04
Epoch 15/50
8736/8736 [==============================] - 1s 128us/sample - loss: 0.0010 - val_loss: 8.9452e-04
Epoch 16/50
8736/8736 [==============================] - 1s 128us/sample - loss: 0.0010 - val_loss: 8.9083e-04
Epoch 17/50
8736/8736 [==============================] - 2s 204us/sample - loss: 0.0010 - val_loss: 8.8754e-04
Epoch 18/50
8736/8736 [==============================] - 1s 129us/sample - loss: 9.9745e-04 - val_loss: 8.8467
e-04
Epoch 19/50
8736/8736 [==============================] - 1s 141us/sample - loss: 9.9401e-04 - val_loss: 8.8210
e-04
Epoch 20/50
8736/8736 [==============================] - 2s 199us/sample - loss: 9.9081e-04 - val_loss: 8.7981
e-04
Epoch 21/50
8736/8736 [==============================] - 2s 182us/sample - loss: 9.8781e-04 - val_loss: 8.7773
e-04
Epoch 22/50
8736/8736 [==============================] - 1s 139us/sample - loss: 9.8498e-04 - val_loss: 8.7584
e-04
Epoch 23/50
8736/8736 [==============================] - 1s 159us/sample - loss: 9.8229e-04 - val_loss: 8.7410
e-04
Epoch 24/50
8736/8736 [==============================] - 2s 180us/sample - loss: 9.7973e-04 - val_loss: 8.7250
e-04
Epoch 25/50
8736/8736 [==============================] - 1s 132us/sample - loss: 9.7729e-04 - val_loss: 8.7100
e-04
Epoch 26/50
8736/8736 [==============================] - 2s 190us/sample - loss: 9.7495e-04 - val_loss: 8.6960
```

```
e-04
Epoch 27/50
8736/8736 [==============================] - 1s 130us/sample - loss: 9.7270e-04 - val_loss: 8.6826
e-04
Epoch 28/50
8736/8736 [==============================] - 1s 129us/sample - loss: 9.7053e-04 - val_loss: 8.6700
e-04
Epoch 29/50
8736/8736 [==============================] - 1s 129us/sample - loss: 9.6843e-04 - val_loss: 8.6578
e-04
Epoch 30/50
8736/8736 [==============================] - 1s 135us/sample - loss: 9.6641e-04 - val_loss: 8.6460
e-04
Epoch 31/50
8736/8736 [==============================] - 1s 132us/sample - loss: 9.6444e-04 - val_loss: 8.6345
e-04
Epoch 32/50
8736/8736 [==============================] - 1s 130us/sample - loss: 9.6253e-04 - val_loss: 8.6233
e-04
Epoch 33/50
8736/8736 [==============================] - 1s 131us/sample - loss: 9.6068e-04 - val_loss: 8.6122
e-04
Epoch 34/50
8736/8736 [==============================] - 1s 151us/sample - loss: 9.5887e-04 - val_loss: 8.6012
e-04
Epoch 35/50
8736/8736 [==============================] - 1s 142us/sample - loss: 9.5711e-04 - val_loss: 8.5904
e-04
Epoch 36/50
8736/8736 [==============================] - 1s 129us/sample - loss: 9.5539e-04 - val_loss: 8.5796
e-04
Epoch 37/50
8736/8736 [==============================] - 1s 127us/sample - loss: 9.5371e-04 - val_loss: 8.5688
e-04
Epoch 38/50
8736/8736 [==============================] - 1s 128us/sample - loss: 9.5206e-04 - val_loss: 8.5581
e-04
Epoch 39/50
8736/8736 [==============================] - 1s 129us/sample - loss: 9.5046e-04 - val_loss: 8.5474
e-04
Epoch 40/50
8736/8736 [==============================] - 1s 129us/sample - loss: 9.4888e-04 - val_loss: 8.5368
e-04
Epoch 41/50
8736/8736 [==============================] - 1s 128us/sample - loss: 9.4735e-04 - val_loss: 8.5263
e-04
Epoch 42/50
8736/8736 [==============================] - 1s 131us/sample - loss: 9.4584e-04 - val_loss: 8.5158
e-04
Epoch 43/50
8736/8736 [==============================] - 1s 129us/sample - loss: 9.4436e-04 - val_loss: 8.5054
e-04
Epoch 44/50
8736/8736 [==============================] - 1s 131us/sample - loss: 9.4292e-04 - val_loss: 8.4951
e-04
Epoch 45/50
8736/8736 [==============================] - 1s 131us/sample - loss: 9.4150e-04 - val_loss: 8.4849
e-04
Epoch 46/50
8736/8736 [==============================] - 1s 131us/sample - loss: 9.4011e-04 - val_loss: 8.4747
e-04
Epoch 47/50
8736/8736 [==============================] - 1s 129us/sample - loss: 9.3874e-04 - val_loss: 8.4645
e-04
Epoch 48/50
8736/8736 [==============================] - 1s 156us/sample - loss: 9.3740e-04 - val_loss: 8.4544
e-04
Epoch 49/50
8736/8736 [==============================] - 1s 145us/sample - loss: 9.3609e-04 - val_loss: 8.4444
e-04
Epoch 50/50
8736/8736 [==============================] - 1s 135us/sample - loss: 9.3479e-04 - val_loss: 8.4345
e-04
```

In [56]:

```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_2 (LSTM)                (None, 100)               50000
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 101
=================================================================
Total params: 50,101
Trainable params: 50,101
Non-trainable params: 0
_________________________________________________________________
```

3.2.3.2 Evaluating Model

In [57]:

```
# predicting
prediction = {
    'train_predict': model.predict(data['train_in']),
    'test_predict': model.predict(data['test_in'])
}

# invert prediction
prediction['train_predict'] = scaler.inverse_transform(prediction['train_predict'])
data['train_out'] = scaler.inverse_transform([data['train_out']])
prediction['test_predict'] = scaler.inverse_transform(prediction['test_predict'])
data['test_out'] = scaler.inverse_transform([data['test_out']])
```

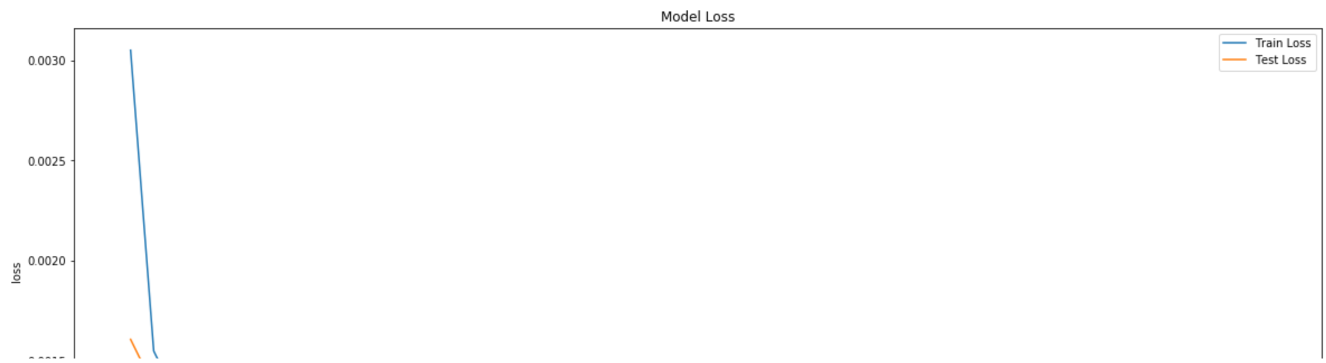
In [58]:

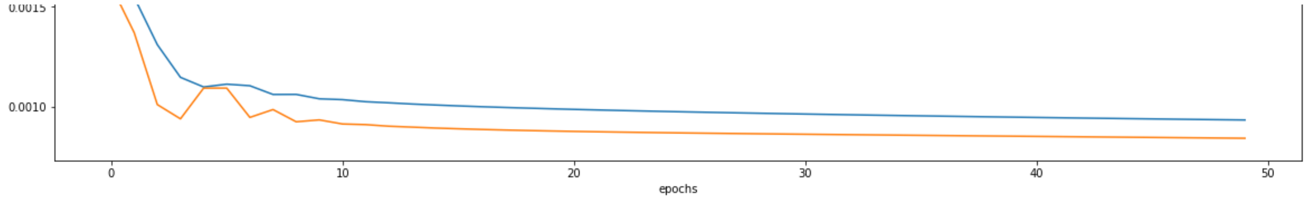
```
# evaluate performance
print('Train MAE:', mean_absolute_error(data['train_out'][0], prediction['train_predict'][:,0]))
print('Train RMSE:',np.sqrt(mean_squared_error(data['train_out'][0], prediction['train_predict']
[:,0])))
print('Test MAE:', mean_absolute_error(data['test_out'][0], prediction['test_predict'][:,0]))
print('Test RMSE:',np.sqrt(mean_squared_error(data['test_out'][0], prediction['test_predict']
[:,0])))
```

```
Train MAE: 16.506829979319907
Train RMSE: 31.54890737093336
Test MAE: 15.824654052060763
Test RMSE: 28.867913903920243
```

In [59]:

```
# plotting loss
plt.figure(figsize=(20,8))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Test Loss')
plt.title('Model Loss')
plt.ylabel('loss')
plt.xlabel('epochs')
plt.legend(loc='upper right')
plt.show();
```





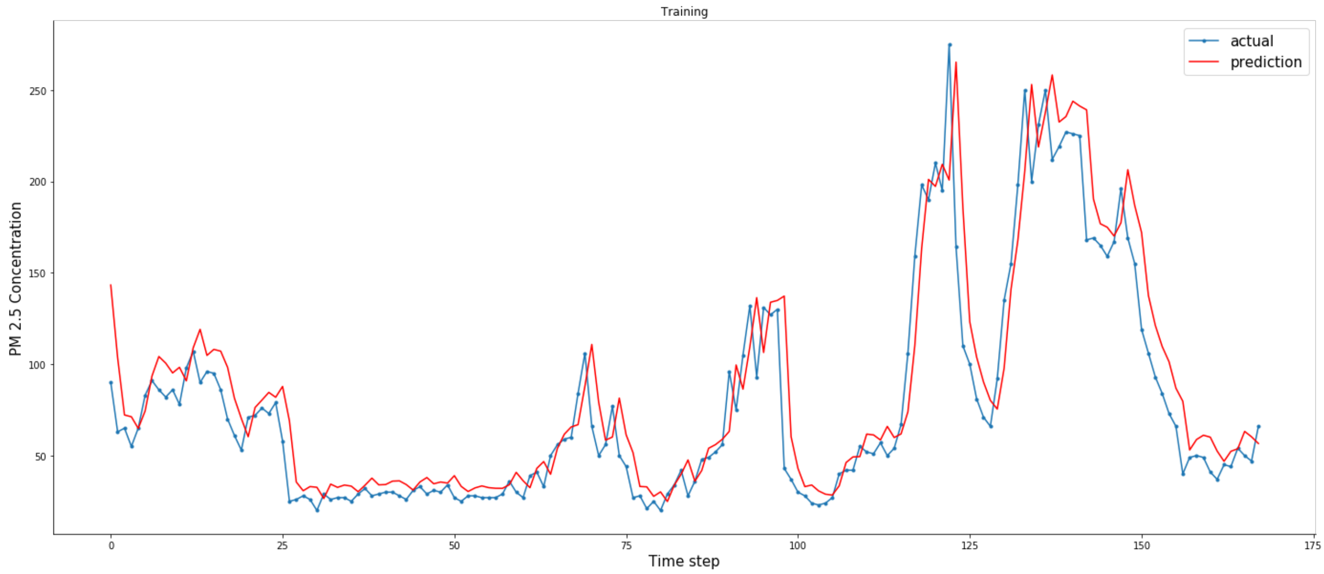
This kind of graph shows that the model is **a good fit**. A good fit is a case where the performance of the model is good on both the train and validation sets. This can be diagnosed from a plot where the train and validation loss decrease and stabilize around the same point.

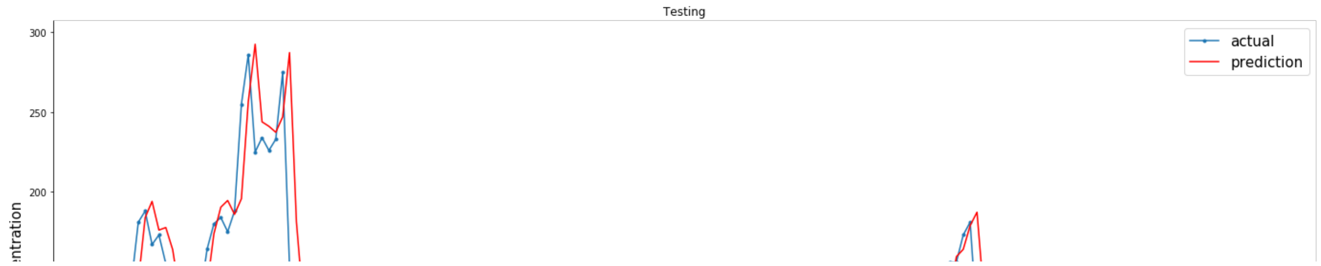
In [60]:

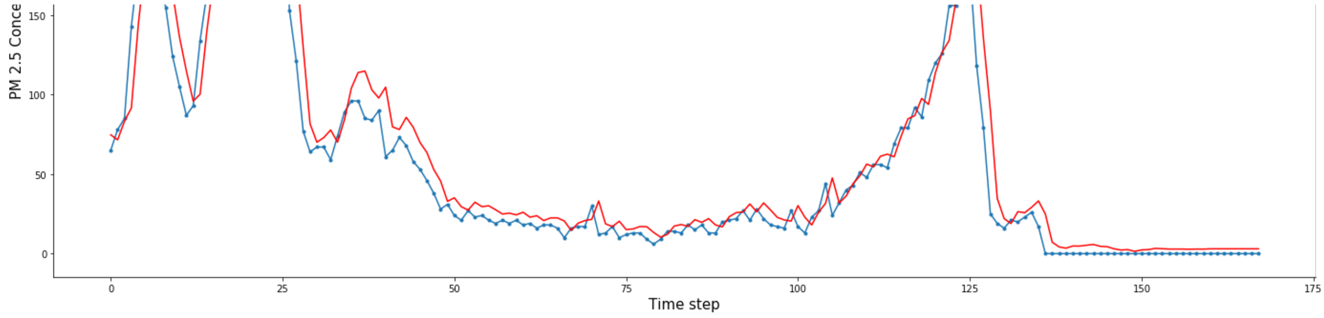
```
# show actuals and predictions on training and testing data in the first week
rng = 24*7
aa = [x for x in range(rng)]

plt.figure(figsize=(20,8))
plt.plot(aa, data['train_out'][0][:rng], marker='.', label="actual")
plt.plot(aa, prediction['train_predict'][:,0][:rng], 'r', label="prediction")
plt.tight_layout()
sns.despine(top=True)
plt.subplots_adjust(left=0.07)
plt.title('Training')
plt.ylabel('PM 2.5 Concentration', size=15)
plt.xlabel('Time step', size=15)
plt.legend(fontsize=15)
plt.show();

plt.figure(figsize=(20,8))
plt.plot(aa, data['test_out'][0][:rng], marker='.', label="actual")
plt.plot(aa, prediction['test_predict'][:,0][:rng], 'r', label="prediction")
plt.tight_layout()
sns.despine(top=True)
plt.subplots_adjust(left=0.07)
plt.title('Testing')
plt.ylabel('PM 2.5 Concentration', size=15)
plt.xlabel('Time step', size=15)
plt.legend(fontsize=15)
plt.show();
```







3.2.4 CNN Model

- Conv1D
- MaxPooling1D
- Flatten
- Dense

In [73]:

```
# reshape input menjadi 3D array : [samples, time steps, features]
data['train_in'] = np.reshape(data['train_in'], (data['train_in'].shape[0], data['train_in'] .shape
[1], 1))
data['test_in']  = np.reshape(data['test_in'], (data['test_in'].shape[0],
data['test_in'].shape[1], 1))
```

In [74]:

```
# design model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(24, 1)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

3.2.4.1 Training

In [76]:

```
# train data and recorded in history
history = model.fit(data['train_in'], data['train_out'], epochs=50, batch_size=70,
                    validation_data=(data['test_in'], data['test_out']), verbose=1, shuffle=False)
```

```
Train on 8736 samples, validate on 35016 samples
Epoch 1/50
8736/8736 [==============================] - 1s 128us/sample - loss: 0.0019 - val_loss: 0.0027
Epoch 2/50
8736/8736 [==============================] - 1s 120us/sample - loss: 0.0018 - val_loss: 0.0030
Epoch 3/50
8736/8736 [==============================] - 1s 130us/sample - loss: 0.0019 - val_loss: 0.0046
Epoch 4/50
8736/8736 [==============================] - 1s 121us/sample - loss: 0.0021 - val_loss: 0.0035
Epoch 5/50
8736/8736 [==============================] - 1s 123us/sample - loss: 0.0021 - val_loss: 0.0031
Epoch 6/50
8736/8736 [==============================] - 1s 126us/sample - loss: 0.0018 - val_loss: 0.0038
Epoch 7/50
8736/8736 [==============================] - 1s 132us/sample - loss: 0.0020 - val_loss: 0.0015
Epoch 8/50
8736/8736 [==============================] - 1s 128us/sample - loss: 0.0017 - val_loss: 0.0016
Epoch 9/50
8736/8736 [==============================] - 1s 124us/sample - loss: 0.0017 - val_loss: 0.0015
Epoch 10/50
8736/8736 [==============================] - 1s 126us/sample - loss: 0.0017 - val_loss: 0.0015
Epoch 11/50
8736/8736 [==============================] - 1s 120us/sample - loss: 0.0017 - val_loss: 0.0015
```

```
Epoch 12/50
8736/8736 [==============================] - 1s 121us/sample - loss: 0.0016 - val_loss: 0.0015
Epoch 13/50
8736/8736 [==============================] - 1s 138us/sample - loss: 0.0016 - val_loss: 0.0015
Epoch 14/50
8736/8736 [==============================] - 1s 132us/sample - loss: 0.0016 - val_loss: 0.0014
Epoch 15/50
8736/8736 [==============================] - 2s 180us/sample - loss: 0.0016 - val_loss: 0.0015
Epoch 16/50
8736/8736 [==============================] - 1s 124us/sample - loss: 0.0016 - val_loss: 0.0015
Epoch 17/50
8736/8736 [==============================] - 1s 122us/sample - loss: 0.0016 - val_loss: 0.0015
Epoch 18/50
8736/8736 [==============================] - 1s 122us/sample - loss: 0.0016 - val_loss: 0.0014
Epoch 19/50
8736/8736 [==============================] - 1s 121us/sample - loss: 0.0016 - val_loss: 0.0015
Epoch 20/50
8736/8736 [==============================] - 1s 124us/sample - loss: 0.0016 - val_loss: 0.0014
Epoch 21/50
8736/8736 [==============================] - 1s 123us/sample - loss: 0.0016 - val_loss: 0.0015
Epoch 22/50
8736/8736 [==============================] - 1s 124us/sample - loss: 0.0016 - val_loss: 0.0014
Epoch 23/50
8736/8736 [==============================] - 1s 124us/sample - loss: 0.0016 - val_loss: 0.0014
Epoch 24/50
8736/8736 [==============================] - 1s 125us/sample - loss: 0.0015 - val_loss: 0.0014
Epoch 25/50
8736/8736 [==============================] - 1s 126us/sample - loss: 0.0015 - val_loss: 0.0014
Epoch 26/50
8736/8736 [==============================] - 1s 125us/sample - loss: 0.0015 - val_loss: 0.0014
Epoch 27/50
8736/8736 [==============================] - 1s 137us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 28/50
8736/8736 [==============================] - 1s 134us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 29/50
8736/8736 [==============================] - 1s 126us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 30/50
8736/8736 [==============================] - 1s 130us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 31/50
8736/8736 [==============================] - 1s 124us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 32/50
8736/8736 [==============================] - 1s 127us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 33/50
8736/8736 [==============================] - 1s 126us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 34/50
8736/8736 [==============================] - 1s 125us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 35/50
8736/8736 [==============================] - 2s 176us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 36/50
8736/8736 [==============================] - 1s 144us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 37/50
8736/8736 [==============================] - 1s 125us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 38/50
8736/8736 [==============================] - 1s 124us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 39/50
8736/8736 [==============================] - 1s 123us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 40/50
8736/8736 [==============================] - 1s 124us/sample - loss: 0.0015 - val_loss: 0.0015
Epoch 41/50
8736/8736 [==============================] - 1s 126us/sample - loss: 0.0014 - val_loss: 0.0015
Epoch 42/50
8736/8736 [==============================] - 1s 126us/sample - loss: 0.0014 - val_loss: 0.0015
Epoch 43/50
8736/8736 [==============================] - 1s 127us/sample - loss: 0.0014 - val_loss: 0.0015
Epoch 44/50
8736/8736 [==============================] - 1s 126us/sample - loss: 0.0014 - val_loss: 0.0015
Epoch 45/50
8736/8736 [==============================] - 1s 127us/sample - loss: 0.0014 - val_loss: 0.0015
Epoch 46/50
8736/8736 [==============================] - 1s 148us/sample - loss: 0.0014 - val_loss: 0.0015
Epoch 47/50
8736/8736 [==============================] - 1s 126us/sample - loss: 0.0014 - val_loss: 0.0015
Epoch 48/50
8736/8736 [==============================] - 1s 126us/sample - loss: 0.0014 - val_loss: 0.0015
Epoch 49/50
8736/8736 [==============================] - 1s 125us/sample - loss: 0.0014 - val_loss: 0.0015
Epoch 50/50
```

```
8736/8736 [==============================] - 1s 126us/sample - loss: 0.0014 - val_loss: 0.0015
```

In [77]:

```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d (Conv1D)              (None, 23, 64)            192
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 11, 64)            0
_________________________________________________________________
flatten (Flatten)            (None, 704)               0
_________________________________________________________________
dense_3 (Dense)              (None, 50)                35250
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 51
=================================================================
Total params: 35,493
Trainable params: 35,493
Non-trainable params: 0
_________________________________________________________________
```

3.2.4.2 Evaluating Model

In [78]:

```
# predicting
prediction = {
    'train_predict': model.predict(data['train_in']),
    'test_predict': model.predict(data['test_in'])
}

# invert prediction
prediction['train_predict'] = scaler.inverse_transform(prediction['train_predict'])
data['train_out'] = scaler.inverse_transform([data['train_out']])
prediction['test_predict'] = scaler.inverse_transform(prediction['test_predict'])
data['test_out'] = scaler.inverse_transform([data['test_out']])
```

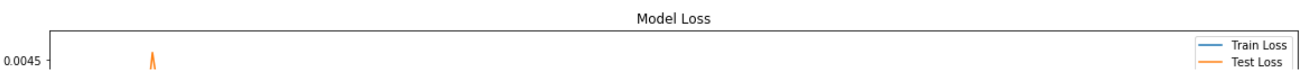
In [79]:

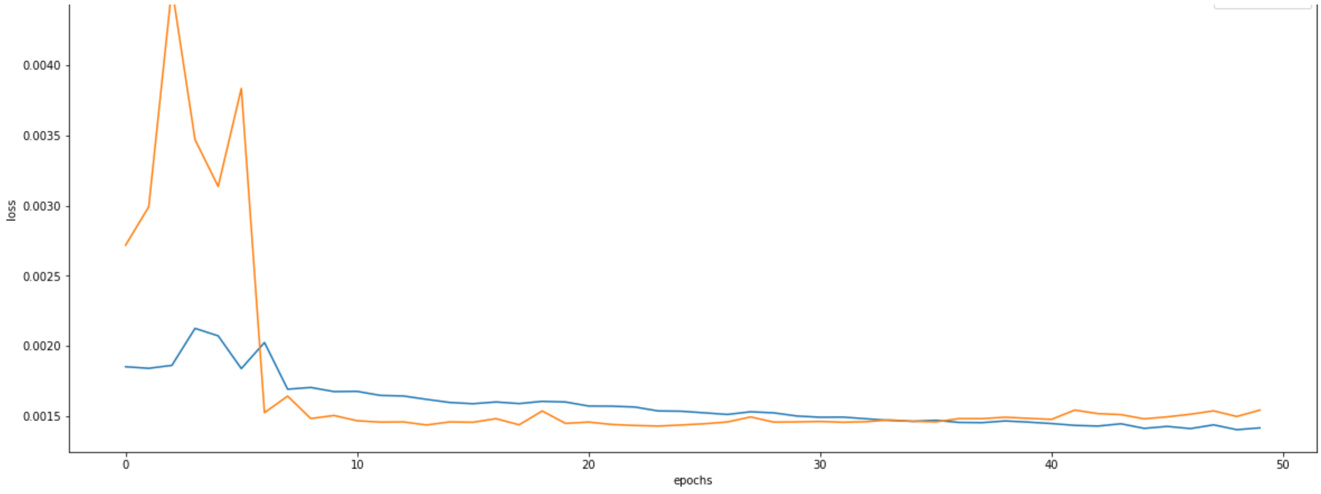
```
# evaluate performance
print('Train MAE:', mean_absolute_error(data['train_out'][0], prediction['train_predict'][:,0]))
print('Train RMSE:',np.sqrt(mean_squared_error(data['train_out'][0], prediction['train_predict']
[:,0])))
print('Test MAE:', mean_absolute_error(data['test_out'][0], prediction['test_predict'][:,0]))
print('Test RMSE:',np.sqrt(mean_squared_error(data['test_out'][0], prediction['test_predict']
[:,0])))
```

```
Train MAE: 22.74579175460517
Train RMSE: 37.475071563818
Test MAE: 22.922563535337922
Test RMSE: 39.04075381532723
```

In [80]:

```
# plotting loss
plt.figure(figsize=(20,8))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Test Loss')
plt.title('Model Loss')
plt.ylabel('loss')
plt.xlabel('epochs')
plt.legend(loc='upper right')
plt.show();
```





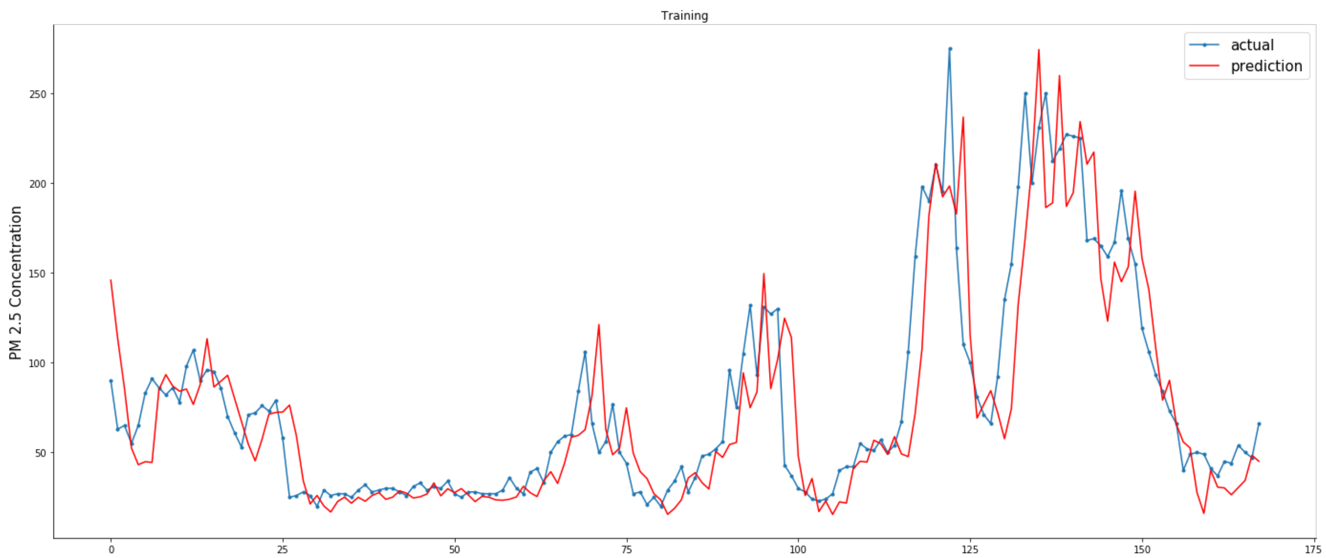
This kind of graph shows that the model is **an overfit**. An overfit model is one where performance on the train set is good and continues to improve, whereas performance on the validation set improves to a point and then begins to degrade. This can be diagnosed from a plot where the train loss slopes down and the validation loss slopes down, hits an inflection point, and starts to slope up again.

In [81]:

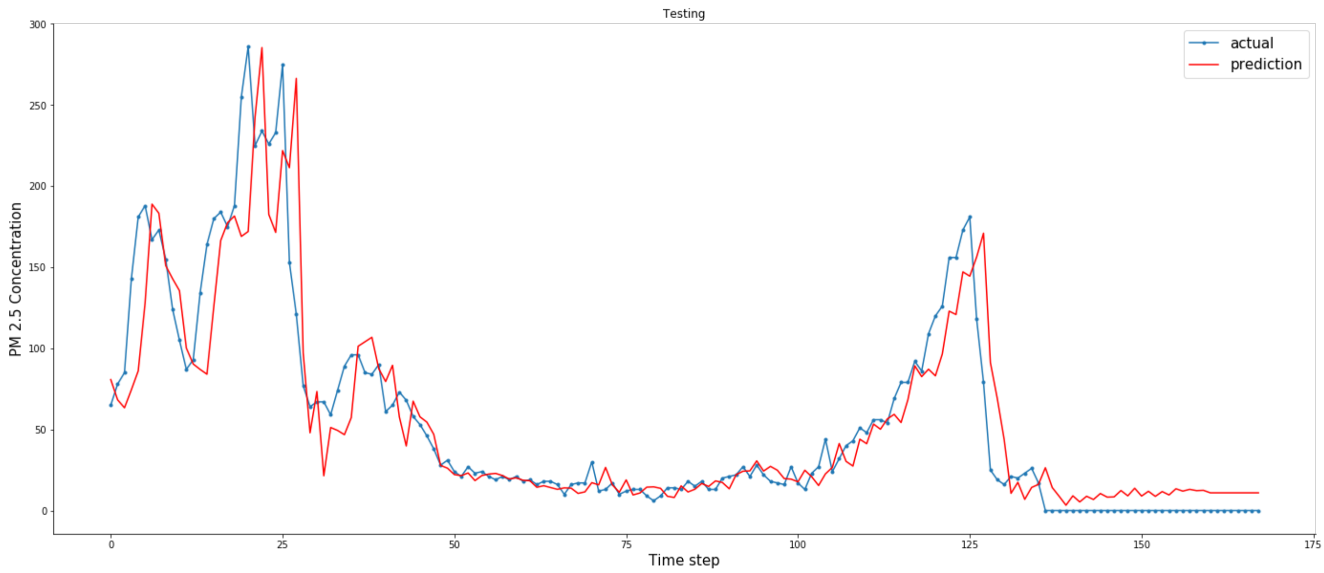
```
# show actuals and predictions on training and testing data in the first week
rng = 24*7
aa = [x for x in range(rng)]

plt.figure(figsize=(20,8))
plt.plot(aa, data['train_out'][0][:rng], marker='.', label="actual")
plt.plot(aa, prediction['train_predict'][:,0][:rng], 'r', label="prediction")
plt.tight_layout()
sns.despine(top=True)
plt.subplots_adjust(left=0.07)
plt.title('Training')
plt.ylabel('PM 2.5 Concentration', size=15)
plt.xlabel('Time step', size=15)
plt.legend(fontsize=15)
plt.show();

plt.figure(figsize=(20,8))
plt.plot(aa, data['test_out'][0][:rng], marker='.', label="actual")
plt.plot(aa, prediction['test_predict'][:,0][:rng], 'r', label="prediction")
plt.tight_layout()
sns.despine(top=True)
plt.subplots_adjust(left=0.07)
plt.title('Testing')
plt.ylabel('PM 2.5 Concentration', size=15)
plt.xlabel('Time step', size=15)
plt.legend(fontsize=15)
plt.show();
```



Time step



3.2.5 Experiment Result

Based on the evaluation of two models, I obtain the following result:

1. LSTM Model

Experiment	MAE	RMSE
1	15.64	28.75
2	15.76	28.82
3	15.92	28.94
Mean	15.77	28.83

2. CNN Model

Experiment	MAE	RMSE
1	21.86	38.25
2	22.63	39.13
3	23.80	40.31
Mean	22.76	39.23

Metrics	LSTM	CNN
MAE	15.77	22.76
RMSE	28.83	39.23

LSTM has better results in MAE and RMSE metrics than CNN for predictions. However, it is very interesting to know how it performs if the CNN and LSTM models are combined.

4. Full Codes

The full code can be found here.

4.1 How to Run

- Preprocess data

```
python3 main.py 1
```

- Forecast data

```
python3 main.py 2 lstm
python3 main.py 2 cnn
```

5. References

- C.-J. Huang and P.-H. Kuo, "A Deep CNN-LSTM Model for Particulate Matter (PM2.5) Forecasting in Smart Cities," Sensors, vol. 18, p. 2220, 2018.
- D. Aha, "UCI Machine Learning Repository," 19 01 2017. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Beijing+PM2.5+Data. [Accessed 19 11 2019].
- https://machinelearningmastery.com/how-to-develop-convolutional-neural-network-models-for-time-series-forecasting/

In [ ]: