Guidance Note - Use of LSTM Code

Issac Pang

April 12, 2023

This guidance note provides a brief summary of how to use the developed LSTM code for testing. In general, the LSTM code consists of two main modules: **Training** and **Utility Functions**. The **Training** module contains all LSTM algorithms (e.g., LSTM(Vanilla) and TCN). If you wish to use/develop the LSTM algorithms to train a new model, please go to the **Training** module. The **Utility Functions** module is mainly used to generate the 3D CSMIP data for training and to evaluate the performance of trained LSTM models.

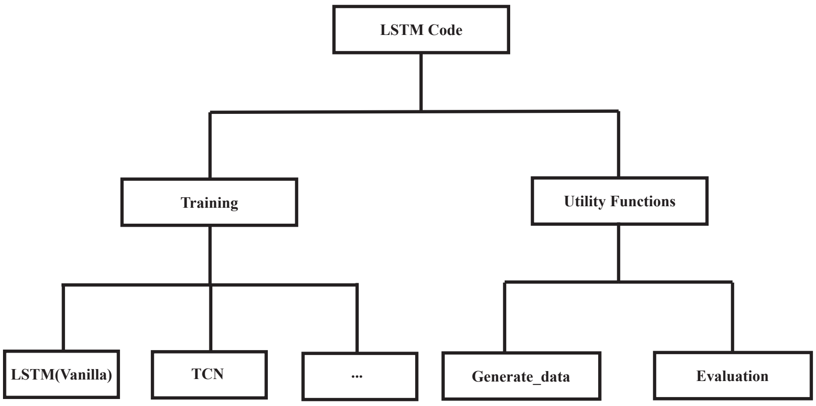

Figure 1: LSTM Code Organization

To use the code for testing or for other research purposes, please download all files in the **Training** and **Utility Functions** modules. Then, update the directory of **train__folder__path** (line 9) and **modelPath** (line 39). The **train__folder__path** is used to store all CSMIP training data, and the **modelPath** is used to save the trained LSTM model. Number of time steps may also need to be updated (line 10), depending on the CSMIP data.

Example: LSTM(vanilla).py

```
1  ...
2  
3  if __name__ == "__main__":
4  
5      #specify the train_folder_path
6      #specify time step (each seismic event has different timesteps, so need to define manually)
7      #specify output_response (e.g. Accel data in Channel 3, 5, 8 => then 3)
8      #specify window size (stack size, please refer to Zhang2019 for details)
9      train_folder_path = r'C:\Users\BRACE2\Desktop\CSMIP\data\training'
10     time_step = 7200
11     output_response = 3
12     window_size = 2
13 
14     #CSMIP data is read into "train_file".
15     train_file = Read_CSMIP_data.Read_CSMIP_data(train_folder_path, time_step, output_response,
       ↪  window_size)
16 
17     #The required 3d array will be generated for training
18     datax, datay = train_file.generate_3d_array()
19 
20     _, n_step, featurex = datax.shape
21     _, n_step, featurey = datay.shape
22 
23     print(featurex)
24     print(featurey)
25 
```

```
    print(n_step)

    n_unit = 30
    n_epoch = 10000
    min_lr = 1e-5 #1e-5

    inputs = Input(shape=(n_step, featurex))
    #print (inputs)
    outputs = TimeDistributed(Dense(featurey))(LSTM(n_unit, return_sequences=True)(inputs))
    print (outputs)

    model1 = Model(inputs=inputs, outputs=outputs)

    modelPath = r'C:\Users\BRACE2\Desktop\CSMIP\model\\'
    mcp_save = ModelCheckpoint(modelPath+ "vanilla_" +str(n_unit)+".hdf5", save_best_only=True, monitor='loss', mode='min')
    reduce_lr_loss = ReduceLROnPlateau(monitor='loss', factor=0.9, patience=500, verbose=1, mode='min', min_lr=min_lr) #1e-5 50
    early_stopping = EarlyStopping(monitor='loss', patience=20000, verbose=False, restore_best_weights=True) #2000

    #adam = optimizers.Adam(lr=1e-5)
    model1.compile(optimizer='adam', loss='mean_squared_error')
    starttime = time.time()
    # history = model1.fit(datax, datay, shuffle=True, epochs=n_epoch, verbose=1, callbacks=[mcp_save, reduce_lr_loss, early_stopping])

    history = model1.fit(datax, datay, shuffle=True, epochs=n_epoch, verbose=1)

    runtime = time.time()-starttime

    dictionary = {}
    dictionary['model'] = model1
    dictionary['history'] = history
    dictionary['runtime'] = runtime
    save_model_dict(dictionary, "vanilla_"+str(n_unit)+"_addFC", modelPath)
```

After training the LSTM model, **Evaluation.py** can be used to evaluate the accuracy of the prediction. Please update the directory of **train_folder_path** (line 32), **test_folder_path** (line 33) and **modelPath** (line 48). The **train_folder_path** and **test_folder_path** are used to store all CSMIP training and testing data respectively, and the **modelPath** is used to load the trained LSTM model. Please also update the name of the trained LSTM model (e.g.,"vanilla_30_addFC", no need to include "_model.h5"). The number of time steps may also need to be updated (line 10), depending on the CSMIP data. Lines 60-92 are used to plot the acceleration responses of the predicted results and the target results for both training and testing sets. The "3" in lines 62, 70, 79 and 87 indicates that the index of the interested CSMIP data (it should be the third item in the **train_folder_path** and **test_folder_path**). If required, please change the number to plot the interested CSMIP dataset. Lines 99-104 are used to calculate the correlation coefficient of the predicted and the target results. It should be noted that $np.corrcoef()$ generates a 2×2 coefficient matrix, which shows 1.0 in both diagonal elements, indicating that each dataset is perfectly correlated with itself, and ≤ 1.0 in the off-diagonal elements, indicating that how the two arrays are correlated with each other. The code (lines 99-104) automatically extracts the off-diagonal element, showing the correlation between the predicted and the target results. For the details of calculation, please go to: https://numpy.org/doc/stable/reference/generated/numpy.corrcoef.html

Example: Evaluation.py

```
1  ...
2  def save_model_dict(dictionary, name, modelPath):
3      dictionary['model'].save(modelPath+r"model_response\\"+name+"_model.h5")
4      f = open(modelPath+r"model_response\\"+name+"_history.pkl","wb")
5      pickle.dump(dictionary['history'].history, f)
6      f.close()
7      f = open("runtime.txt", "a")
8      f.write(name+" runtime: ")
9      f.write(str(dictionary['runtime']/60))
```

```
    f.write("\n")
    f.close()

def load_model_dict(name, modelPath):
    dictionary = {}
    try:
        model = load_model(modelPath+r"model_response\\"+name+".hdf5")
    except:
        model = load_model(modelPath+r"model_response\\"+name+"_model.h5")
    dictionary['model'] = model
    f = open(modelPath+r"model_response\\"+name+"_history.pkl", 'rb')
    history = pickle.load(f)
    f.close()
    dictionary['history'] = history
    return dictionary

if __name__ == "__main__":

    #specify the train_folder_path and the test_folder_path
    #specify time step (each seismic event has different timesteps, so need to define manually)
    #specify output_response (e.g. Accel data in Channel 3, 5, 8 => then 3)
    #specify window size (stack size, please refer to Zhang2019 for details)
    train_folder_path = r'C:\Users\BRACE2\Desktop\CSMIP\data\training'
    test_folder_path = r'C:\Users\BRACE2\Desktop\CSMIP\data\testing'

    time_step = 7200
    output_response = 3
    window_size = 2

    #CSMIP data is read into "train_file" and "test_file".
    train_file = Read_CSMIP_data.Read_CSMIP_data(train_folder_path, time_step, output_response,
    ↪  window_size)
    test_file = Read_CSMIP_data.Read_CSMIP_data(test_folder_path, time_step, output_response,
    ↪  window_size)

    #The required 3d array will be generated for training/testing
    datax, datay = train_file.generate_3d_array()
    testx, testy = test_file.generate_3d_array()

    #load the trained model
    modelPath = r'C:\Users\BRACE2\Desktop\CSMIP\model\\'
    model_dict = load_model_dict("vanilla_30_addFC", modelPath)
    model = model_dict['model']

    #perform prediction and load the results in datapredict and testpredict
    datapredict = model.predict(datax)
    testpredict = model.predict(testx)
    print("train_loss:")
    print(model.evaluate(datax, datay, verbose=0))
    print("test_loss:")
    print(model.evaluate(testx, testy, verbose=0))

    #Sample Plot of training data
    plt.figure()
    plt.plot(model.predict(datax)[3,:,0], color='blue', lw=1.0)
    plt.plot(datay[3,:,0],':', color='red', alpha=0.8, lw=1.0)
    plt.title('Training Set: 3rd Floor Acceleration (x-direction)')
    plt.legend(["Predicted", "Real"])
    plt.xlabel("Time Step")
    plt.ylabel("Acceleration (cm/sec$^2$)")

    plt.figure()
    plt.plot(model.predict(datax)[3,:,1], color='blue',lw=1.0)
```

```
    plt.plot(datay[3,:,1],':', color='red', alpha=0.8, lw=1.0)
    plt.title('Training Set: Roof Acceleration (x-direction)')
    plt.legend(["Predicted", "Real"])
    plt.xlabel("Time Step")
    plt.ylabel("Acceleration (cm/sec$^2$)")

    #Sample Plot of testing data
    plt.figure()
    plt.plot(model.predict(testx)[3,:,0], color='blue', lw=1.0)
    plt.plot(testy[3,:,1],':', color='red', alpha=0.8, lw=1.0)
    plt.title('Testing Set: 3rd Floor Acceleration (x-direction)')
    plt.legend(["Predicted", "Real"])
    plt.xlabel("Time Step")
    plt.ylabel("Acceleration (cm/sec$^2$)")

    plt.figure()
    plt.plot(model.predict(testx)[3,:,1], color='blue',lw=1.0)
    plt.plot(testy[3,:,0],':', color='red', alpha=0.8, lw=1.0)
    plt.title('Testing Set: Roof Acceleration (x-direction)')
    plt.legend(["Predicted", "Real"])
    plt.xlabel("Time Step")
    plt.ylabel("Acceleration (cm/sec$^2$)")

    # Correlation Coefficient
    # Note: The resulting matrix from np.corrcoef shows this by having 1.0
    # in both diagonal elements, indicating that each array is perfectly correlated
    # with itself, and < 1.0 in the off-diagonal elements, indicating that how the two arrays
    # are correlated with each other.
    print("training corr")
    train_corr = np.corrcoef(datapredict.flatten(), datay.flatten())[0,1]
    print(train_corr)
    print("testing corr")
    test_corr = np.corrcoef(testpredict.flatten(), testy.flatten())[0,1]
    print(test_corr)


    # Error - evaluate the error between the predicted result and the real result
    errors = np.array([])

    x = (datapredict[:,:,0] - datay[:,:,0]) / np.max(np.abs(datay[:,:,0]),
    ↪  axis=1).reshape((-1,1))
    hist = np.histogram(x.flatten(), np.arange(-0.2, 0.201, 0.001))[0]
    errors = np.append(errors, hist)
    x = (datapredict[:,:,1] - datay[:,:,1]) / np.max(np.abs(datay[:,:,1]),
    ↪  axis=1).reshape((-1,1))
    hist = np.histogram(x.flatten(), np.arange(-0.2, 0.201, 0.001))[0]
    errors = np.append(errors, hist)
    x = (testpredict[:,:,0] - testy[:,:,0]) / np.max(np.abs(testy[:,:,0]),
    ↪  axis=1).reshape((-1,1))
    hist = np.histogram(x.flatten(), np.arange(-0.2, 0.201, 0.001))[0]
    errors = np.append(errors, hist)
    x = (testpredict[:,:,1] - testy[:,:,1]) / np.max(np.abs(testy[:,:,1]),
    ↪  axis=1).reshape((-1,1))
    hist = np.histogram(x.flatten(), np.arange(-0.2, 0.201, 0.001))[0]
    errors = np.append(errors, hist)

    errors = errors.reshape((-1, 4, 400))
    np.save("errors_new.npy", errors)
    error = np.load("errors_new.npy")

    print(error.shape)

    # Print the error graph, a better result will lead to an error curve centralized to 0.
```

```
130     plt.figure()
131     plt.plot(np.arange(-20, 20, 0.1), error[0][0] / (np.sum(error[0][0]) * 0.001))
132     plt.plot(np.arange(-20, 20, 0.1), error[0][1] / (np.sum(error[0][1]) * 0.001))
133     plt.legend(["Third floor", "Roof"])
134     plt.xlim(-20,20)
135     plt.xlabel("Normalized Error (%)")
136     plt.ylabel("PDF")
137     plt.title('Training Set')
138 
139     plt.figure()
140     plt.plot(np.arange(-20, 20, 0.1), error[0][2] / (np.sum(error[0][2]) * 0.001))
141     plt.plot(np.arange(-20, 20, 0.1), error[0][3] / (np.sum(error[0][3]) * 0.001))
142     plt.legend(["Third floor", "Roof"])
143     plt.xlim(-20,20)
144     plt.xlabel("Normalized Error (%)")
145     plt.ylabel("PDF")
146     plt.title('Testing Set')
```

Lines 130-146 are used to generate the normalized error curves of different floors. The errors of all the response cases and time steps are collected and the distribution can be plotted. The more the distribution is centered around 0.0, the better the model.