National College of Ireland

A Neural Network Modelling for Soil Moisture Prediction

MSc Research Project
Data Analytics

Shubham Patil
Student ID: X17165768

School of Computing
National College of Ireland

Supervisor: Bahman Honari

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Shubham Patil
Student ID:	X17165768
Programme:	Data Analytics
Year:	2019
Module:	MSc Research Project
Supervisor:	Bahman Honari
Submission Due Date:	12/12/2019
Project Title:	A Neural Network Modelling for Soil Moisture Prediction
Word Count:	XXX
Page Count:	27

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	12th December 2019

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A Neural Network Modelling for Soil Moisture Prediction

Shubham Patil

X17165768

This is Configuration manual will describe implementation procedure in details,

1 Data Collection

The data used for this research acquired from The U.S. Geological Survey of two different sites[1]. This data made up of an hourly time series record of soil temperature, volumetric soil moisture content. This data collected by the U.S. Geological Survey(USGS)[2] in cooperation with the California Department of Water Resources, National Park Service and Pepperwood Preserve at five different locations across Yosemite National Park. At each location, soil probe are installed from their total soil profile data collected. This dataset developed for the understanding of soil relation to climate and it also helps in the contribution of long term hourly time series soil moisture and temperature dataset. From an available set of dataset Dana Meadows, Gin Flat site's dataset is selected to developed to test models. As shown in below table1, details of sites datasets with name of sites, data records range and count of records for each site and in table2, data definition of both sites.

Site No.	Site Name	Data Range	Count of Records
1	Dana Meadows	09/11/2005 to 07/26/2017	100,420
2	Gin Flat	09/01/2005 to 07/26/2017	99,836

Table 1: Details of the sites data

Column Name	Data Type	Description
Date/Time	datetime64	Date wise records for hours
ST_10cm	float64	Soil temperature of depth 10cm in Celsius
ST_36cm	float64	Soil temperature of depth 36cm in Celsius
Soil_VWC10cm	float64	Volumetric soil moisture of 10cm
Soil_VWC36cm	float64	Volumetric soil moisture of 36cm
Soil_TWC	float64	Total water Content of Soil

Table 2: Data Definition of both sites

[1]USGS-Data `https://pubs.usgs.gov/ds/1083/ds1083_tables12-15_17.zip`.

[2]USGS `https://doi.org/10.3133/ds1083`.

Location of study areas shows in Figure1 belows,

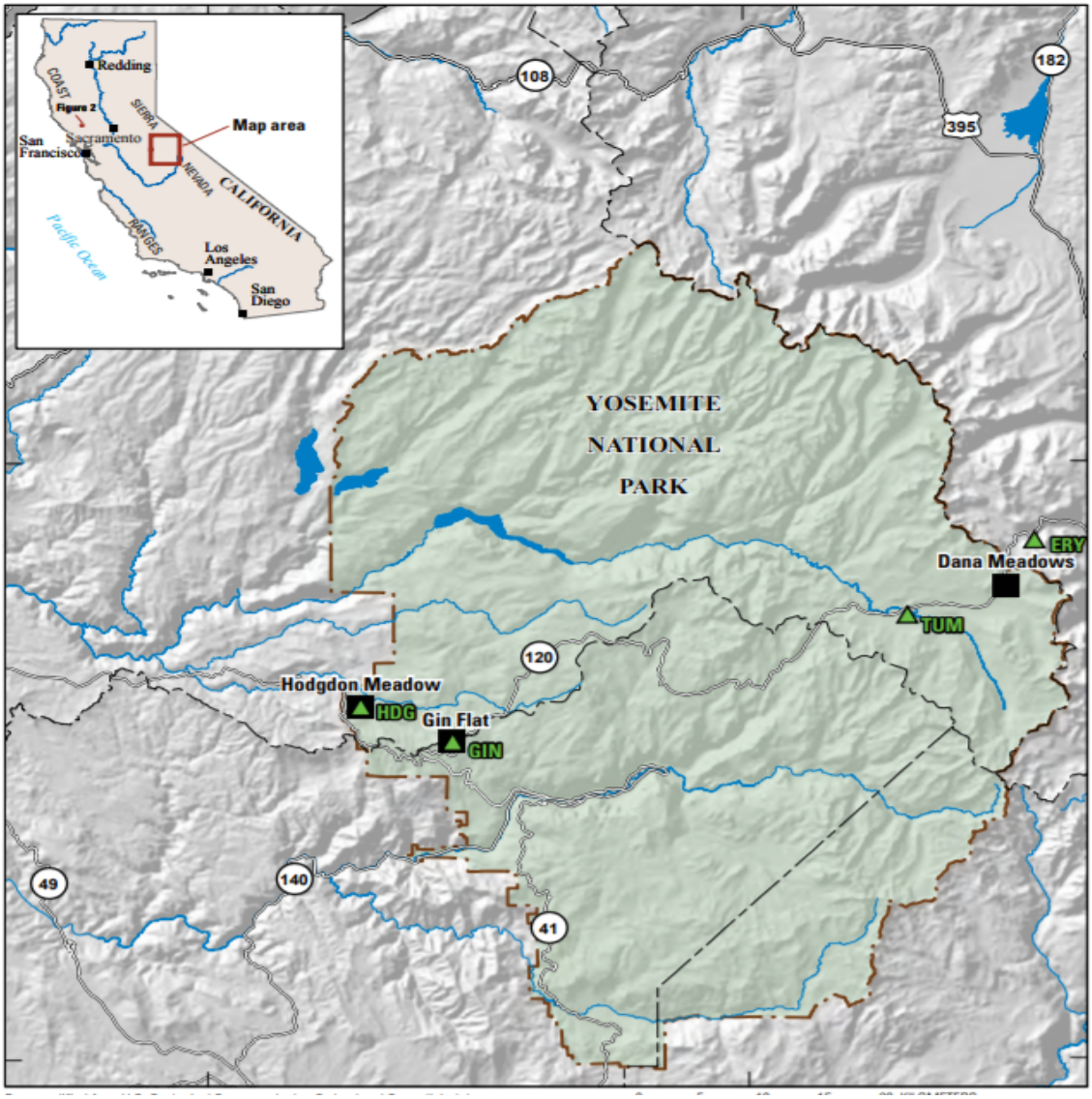

Figure 1: Location of Study Area(Stern et al.; 2018)

2 Software Used

The Software used in this research are explain as below,

- **Python 3.7**: For this study, Python 3.7 used for all neural network models implementation which covers task such as Data Loading, Data Cleaning, Exploratory Data Analysis ,Data Transformation and Neural Network Model development.
- **MS Excel**: A software program developed by Microsoft that allows users to organize, format and calculate data with formulas using spreadsheet system. For this research it used for variables rename right after data acquired.
- **Chrome**: Chrome browser is used to run Google Colab Notebook Session and for Thesis report writing in organized manner on Overleaf website.

- **Overleaf**: Overleaf is web-based academic writing free available service used to write for this thesis report in an organized manner with all required references.

  Microsoft Excel is a software program produced by Microsoft that allows users to organize, format and calculate data with formulas using a spreadsheet system.

3 System Configuration

This research implemented on Google Colab, a freely available cloud based virtual machine environment and in following is it's System Configuration:

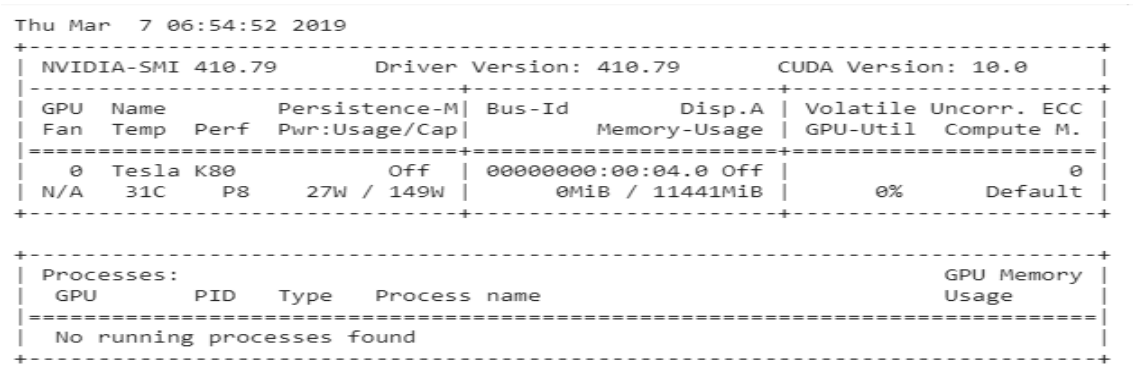

Figure 2: GPU Configuration

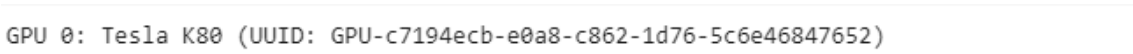

Figure 3: GPU Details

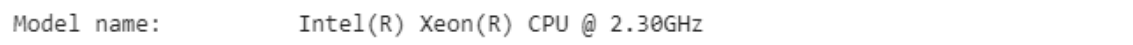

Figure 4: CPU Details

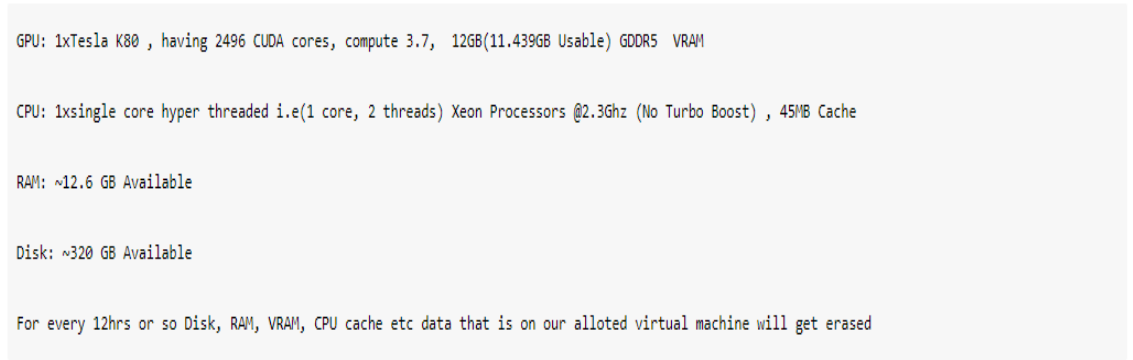

Figure 5: Overall

4 Initial Stage

```
[ ]  #To mount google drive
     from google.colab import drive
     drive.mount('/content/drive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=9473189

Enter your authorization code:
··········
Mounted at /content/drive
```

Figure 6: Files on Google Drive

As two datasets are uploaded to Google Drive, it is essential to connect google drive. Hence at beginning Google Drive is mounted to current Google Colaboratory session using above code.

```
#All libraries
import numpy as np
import pandas as pd
import os
import sys
import io

%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
import os
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import power_transform
from pandas import read_csv
from pandas import DataFrame
from scipy.stats import yeojohnson
from matplotlib import pyplot
from sklearn.preprocessing import PowerTransformer
from sklearn import preprocessing
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Input, LSTM, Dense, GRU, Embedding, Dropout
from tensorflow.python.keras.optimizers import RMSprop
from tensorflow.python.keras.callbacks import EarlyStopping, ModelCheckpoint, TensorBoard, ReduceLROnPlateau
```

Figure 7: Import Libraries

As shown in above figure, list of libraries used while implementing this research in python and TensorFlow is used as backend for research.

```
[ ] #Google drive paths of both sites file
    #address of file is assigen to path variable as per need of particular site

    path="/content/drive/My Drive/DATA/Table13_GinFlat.csv"
    path="/content/drive/My Drive/DATA/Table12_DanaMeadows.csv"
```

```
[ ] #To read required site using required path as mentioned above
    site = pd.read_csv(path)
```

```
[ ] #Change data type of 'Date/Time' Variable as dataetime64 and then set as index
    site['Date/Time']=pd.to_datetime(site['Date/Time'])
    site.set_index(['Date/Time'], inplace=True)
    list(site.columns)
```

Figure 8: Dataset Access

Each site trained and evaluated independently on each given models. Hence 'path' variable used to store the paths of both file but as per need it at each time only one file is assigned to 'path' variable. After that file is read using read_csv() function. The datatype of Date/Time column is changed to datetime64 and set as index of given dataframe.

```
[ ] #for site1
    site=site.drop(columns='ST_53cm')
    site=site.drop(columns='ST_79cm')
    site=site.drop(columns='Soil_WMP10cm')
    site=site.drop(columns='Soil_WMP36cm')
    site=site.drop(columns='Soil_WMP53cm')
    site=site.drop(columns='Soil_WMP79cm')
    site=site.drop(columns='Soil_VWC53cm')
```

```
[ ] #For Site2
    site=site.drop(columns='ST_71cm')
    site=site.drop(columns='Soil_WMP10cm')
    site=site.drop(columns='Soil_WMP36cm')
    site=site.drop(columns='Soil_WMP71cm')
    site=site.drop(columns='Soil_VWC71cm')
```

```
[ ] #change data type to float of all variables in dataframe
    site = site.astype('float')
    site.head()
```

Figure 9: Drop Columns and Change Datatype

This study focused on only 10cm and 36cm depth prediction of volumetric soil moisture. cause of that all other depths variables are dropped. Above figure shows, two different cells for drop variables for two different sites and as per site one of the cell is used. Then to make integrity among variables are converted to float datatype.

```
[ ]  #To check number of missing values in selecected site
     data=site
     np.isnan(data).sum()
```

```
[ ]  #Function for fill missing values
     def fill_missing(data):
         one_day = 24
         for row in range(data.shape[0]):
             for col in range(data.shape[1]):
                 if np.isnan(data[row, col]):
                     data[row, col] = data[row-one_day, col]
```

```
[ ]  #function call to fill missing values of dataframe values
     fill_missing(data.values)
```

```
[ ]  #To check still there are any missing values in dataframe
     np.isnan(data).sum()
```

Figure 10: Missing value Check and Fill

As shown is above a user defined function is used to fill missing values present in dataframe. Firstly, number of missing values are checked. Secondly, then function is called to fill those missing values using previous year values of same day i.e. from previous year data of same day.

5 Exploratory Data Analysis(EDA)

From here exploratory data analysis procedure begin on selected site,

```
[ ]  EDA_df=data
```

```
[ ]  #Statistical Details of Data
     #Max Min Mean Std
     EDA_df.describe()
```

```
[ ]  #Statistical Details of Data
     #Skew & Kurt for each column Start

     print(EDA_df['ST_10cm'].skew())
     print(EDA_df['ST_10cm'].kurt())
```

Figure 11: EDA for Statistical Description

```
[ ] print(EDA_df['ST_36cm'].skew())
    print(EDA_df['ST_36cm'].kurt())

[ ] print(EDA_df['Soil_VWC10cm '].skew())
    print(EDA_df['Soil_VWC10cm '].kurt())

[ ] print(EDA_df['Soil_VWC36cm '].skew())
    print(EDA_df['Soil_VWC36cm '].kurt())

[ ] print(EDA_df['Soil_TWC'].skew())
    print(EDA_df['Soil_TWC'].kurt())
    #Skew & Kurt for each column End
```

Figure 12: EDA for Statistical Description

As shown in above Figure11 and Figure12 code to decribe statistical description of each variables present in dataframe. The new dataframe is defined by assigning previous dataframe to do further analysis. The describe() presents in python is used to get statistical acknowledgement of site such as Mean, Standard deviation, Minimum and Maximum. Furthermore, skew() and kurt() functions of python used to find out skewness and kurtosis for each variables of given dataframe of selected site.

```
[ ] import seaborn as sns
    EDA_df=data
    season = EDA_df
    season['Date'] = EDA_df.index.date
    season['Year'] = EDA_df.index.year
    season['Month'] = EDA_df.index.month

[ ] #Start of Time-Series Seasonality Plot for each Variable
    sns.set()
    spivot = pd.pivot_table(season, index='Month', columns = 'Year', values = 'ST_10cm', aggfunc=np.mean)
    ax=spivot.plot(figsize=(15,5), linewidth=3)
    ax.set_xlabel('Months')
    ax.set_ylabel('Soil Temperature(°C)')
    ax.set_title('Seasonality in Soil Temperature of 10cm depth ');
    #as per site following code change
    #ax.text(7,1,'for the Dana Meadows site', style='italic',bbox={'facecolor': 'aqua', 'alpha': 0.5, 'pad': 10})
    ax.text(7,1,'for the Gin Flat site', style='italic',bbox={'facecolor': 'aqua', 'alpha': 0.5, 'pad': 10})
    plt.show()
```

Figure 13: EDA for Time-Series Seasonality

```
[ ] sns.set()
    spivot = pd.pivot_table(season, index='Month', columns = 'Year', values = 'ST_36cm', aggfunc=np.mean)
    ax=spivot.plot(figsize=(15,5), linewidth=3)
    ax.set_xlabel('Months')
    ax.set_ylabel('Soil Temperature(°C)')
    ax.set_title('Seasonality in Soil Temperature of 36cm depth ');
    #as per site following code change
    #ax.text(7,2,'for the Dana Meadows site', style='italic',bbox={'facecolor': 'aqua', 'alpha': 0.5, 'pad': 10})
    ax.text(7,1,'for the Gin Flat site', style='italic',bbox={'facecolor': 'aqua', 'alpha': 0.5, 'pad': 10})
    plt.show()
```

```
[ ] sns.set()
    spivot = pd.pivot_table(season, index='Month', columns = 'Year', values = 'Soil_VWC10cm ', aggfunc=np.mean)
    ax=spivot.plot(figsize=(15,5), linewidth=3)
    ax.set_xlabel('Months')
    ax.set_ylabel('Soil Volumetric soil water content')
    ax.set_title('Seasonality in Volumetric soil water content 10cm depth(in cubic centimeters per cubic centimeter)');
    #as per site following code change
    #ax.text(8,0.5,'for the Dana Meadows site', style='italic',bbox={'facecolor': 'aqua', 'alpha': 0.5, 'pad': 10})
    ax.text(10,0.25,'for the Gin Flat site', style='italic',bbox={'facecolor': 'aqua', 'alpha': 0.5, 'pad': 10})
    plt.show()
```

Figure 14: EDA for Time-Series Seasonality

```
[ ] sns.set()
    spivot = pd.pivot_table(season, index='Month', columns = 'Year', values = 'Soil_VWC36cm ', aggfunc=np.mean)
    ax=spivot.plot(figsize=(15,5), linewidth=3)
    ax.set_xlabel('Months')
    ax.set_ylabel('Soil Volumetric soil water content')
    ax.set_title('Seasonality in Volumetric soil water content 36cm depth(in cubic centimeters per cubic centimeter)');
    #as per site following code change
    #ax.text(8,0.5,'for the Dana Meadows site', style='italic',bbox={'facecolor': 'aqua', 'alpha': 0.5, 'pad': 10})
    ax.text(4,0.1,'for the Gin Flat site', style='italic',bbox={'facecolor': 'aqua', 'alpha': 0.5, 'pad': 10})
    plt.show()
```

```
[ ] sns.set()
    spivot = pd.pivot_table(season, index='Month', columns = 'Year', values = 'Soil_TWC ', aggfunc=np.mean)
    ax=spivot.plot(figsize=(15,5), linewidth=3)
    ax.set_xlabel('Months')
    ax.set_ylabel('Soil Total water content')
    ax.set_title('Seasonality in Total soil water content');
    #as per site following code change
    #ax.text(8,0.5,'for the Dana Meadows site', style='italic',bbox={'facecolor': 'aqua', 'alpha': 0.5, 'pad': 10})
    ax.text(4,0.1,'for the Gin Flat site', style='italic',bbox={'facecolor': 'aqua', 'alpha': 0.5, 'pad': 10})
    plt.show()
    #End of Time-Series Seasonality Plot for each Variable
```

Figure 15: EDA for Time-Series Seasonality

As shown in above Figure13 to Figure15 seasonality of each variable is find out by plotting graph. Plot describes monthly seasonality for each variable in all the years represents in dataframe. Each year indicated by different color in graph and color palette also plotted, this make each graph eye catching and easy to learn(*Time Series Analysis in Python – A Comprehensive Guide with Examples*; 2019).

```
[ ] #Star of Time-Series Trend Plot for each Variable
    fig, axes = plt.subplots(1, 2, figsize=(20,2), dpi= 80)
    sns.lineplot(x='Year', y='ST_10cm', data=season, ax=axes[0])
    sns.lineplot(x='Year', y='ST_36cm', data=season,ax=axes[1])

    axes[0].set_title('Trend in Soil Temperature of 10cm depth(Year wise)', fontsize=15);
    axes[0].set_xlabel('Year')
    axes[0].set_ylabel('Soil Temperature(°C)')

    axes[1].set_title('Trend in Soil Temperature of 36cm depth(Year wise)', fontsize=15);
    axes[1].set_xlabel('Year')
    axes[1].set_ylabel('Soil Temperature(°C)')


    plt.show()
```

Figure 16: EDA for Time-Series Trend

```
[ ]
    fig, axes = plt.subplots(1, 2, figsize=(20,2), dpi= 80)
    sns.lineplot(x='Year', y='Soil_VWC10cm', data=season, ax=axes[0])
    sns.lineplot(x='Year', y='Soil_VWC36cm', data=season,ax=axes[1])

    axes[0].set_title('Trend in Volumetric soil water content 10cm depth(Year wise)', fontsize=15);
    axes[0].set_xlabel('Year')
    axes[0].set_ylabel('volumetric water content')


    axes[1].set_title('Trend in Volumetric soil water content 36cm depth(Year wise)', fontsize=15);
    axes[1].set_xlabel('Year')
    axes[1].set_ylabel('volumetric water content')


    plt.show()
```

Figure 17: EDA for Time-Series Trend

```
[ ] fig, axes = plt.subplots(1, 2, figsize=(20,2), dpi= 80)
    sns.lineplot(x='Year', y='Soil_TWC', data=season, ax=axes[0])

    axes[0].set_title('Trend in Soil total water content 10cm depth(Year wise)', fontsize=15);
    axes[0].set_xlabel('Year')
    axes[0].set_ylabel('total water content')


    plt.show()
    #End of Time-Series Trend Plot for each Variable
```

Figure 18: EDA for Time-Series Trend

As shown in Figure16 to Figure18 trend in each variable is determined. Plot describes

yearly trend for each variable of dataframe of selected site(*Time Series Analysis in Python – A Comprehensive Guide with Examples*; 2019).

```
[ ]  #Start of Test For Stationary Check for each Variable
     from statsmodels.tsa.stattools import adfuller, kpss
     # ADF Test
     result = adfuller(EDA_df.ST_10cm.values, autolag='AIC')
     print(f'ADF Statistic: {result[0]}')
     print(f'p-value: {result[1]}')
     for key, value in result[4].items():
         print('Critial Values:')
         print(f'   {key}, {value}')

     # KPSS Test
     result = kpss(EDA_df.ST_10cm.values, regression='c')
     print('\nKPSS Statistic: %f' % result[0])
     print('p-value: %f' % result[1])
     for key, value in result[3].items():
         print('Critial Values:')
         print(f'   {key}, {value}')
```

Figure 19: EDA for Time-Series Stationary Check

```
[ ]  # ADF Test
     result = adfuller(EDA_df.ST_36cm.values, autolag='AIC')
     print(f'ADF Statistic: {result[0]}')
     print(f'p-value: {result[1]}')
     for key, value in result[4].items():
         print('Critial Values:')
         print(f'   {key}, {value}')

     # KPSS Test
     result = kpss(EDA_df.ST_36cm.values, regression='c')
     print('\nKPSS Statistic: %f' % result[0])
     print('p-value: %f' % result[1])
     for key, value in result[3].items():
         print('Critial Values:')
         print(f'   {key}, {value}')
```

Figure 20: EDA for Time-Series Stationary Check

```
[ ]  # ADF Test
     result = adfuller(EDA_df.Soil_VWC10cm.values, autolag='AIC')
     print(f'ADF Statistic: {result[0]}')
     print(f'p-value: {result[1]}')
     for key, value in result[4].items():
         print('Critial Values:')
         print(f'   {key}, {value}')

     # KPSS Test
     result = kpss(EDA_df.Soil_VWC10cm.values, regression='c')
     print('\nKPSS Statistic: %f' % result[0])
     print('p-value: %f' % result[1])
     for key, value in result[3].items():
         print('Critial Values:')
         print(f'   {key}, {value}')
```

Figure 21: EDA for Time-Series Stationary Check

```
[ ]  # ADF Test
     result = adfuller(EDA_df.Soil_VWC36cm.values, autolag='AIC')
     print(f'ADF Statistic: {result[0]}')
     print(f'p-value: {result[1]}')
     for key, value in result[4].items():
         print('Critial Values:')
         print(f'   {key}, {value}')

     # KPSS Test
     result = kpss(EDA_df.Soil_VWC36cm.values, regression='c')
     print('\nKPSS Statistic: %f' % result[0])
     print('p-value: %f' % result[1])
     for key, value in result[3].items():
         print('Critial Values:')
         print(f'   {key}, {value}')
```

Figure 22: EDA for Time-Series Stationary Check

```
[ ]  # ADF Test
     result = adfuller(EDA_df.Soil_TWC.values, autolag='AIC')
     print(f'ADF Statistic: {result[0]}')
     print(f'p-value: {result[1]}')
     for key, value in result[4].items():
         print('Critial Values:')
         print(f'   {key}, {value}')

     # KPSS Test
     result = kpss(EDA_df.Soil_TWC.values, regression='c')
     print('\nKPSS Statistic: %f' % result[0])
     print('p-value: %f' % result[1])
     for key, value in result[3].items():
         print('Critial Values:')
         print(f'   {key}, {value}')

     #End of Test For Stationary Check for each Variable
```

Figure 23: EDA for Time-Series Stationary Check

As depicted in above Figures19 to Figure23 code of stationary tests of each variable using ADF(Augmented Dickey Fuller) KPSS(Kwiatkowski-Phillips-Schmidt-Shin) tests. Results of each tests shows that given variables are stationary to implement given time series prediction study(SINGH; 2018).

6 Implementation of Models

```
[ ] data.info()
```

```
[ ] #Start of Deseasonalize and Detrend of each variable

    from statsmodels.tsa.seasonal import seasonal_decompose
    from dateutil.parser import parse

    # Additive Decomposition
    result_add = seasonal_decompose(data['ST_10cm'], model='additive', extrapolate_trend='freq',freq=24)
    # Deseasonalize ST_10cm
    data.ST_10cm = data.ST_10cm.values / result_add.seasonal
    #Detrend ST_10cm
    data.ST_10cm = data.ST_10cm.values - result_add.trend

    # Additive Decomposition
    result_add = seasonal_decompose(data['ST_36cm'], model='additive', extrapolate_trend='freq',freq=24)
    # Deseasonalize ST_10cm
    data.ST_36cm = data.ST_36cm.values / result_add.seasonal
    #Detrend ST_10cm
    data.ST_36cm = data.ST_36cm.values - result_add.trend
```

Figure 24: Removal of Seasonality and Trend From Time-Series Data

```
# Additive Decomposition
result_add = seasonal_decompose(data['Soil_VWC10cm'], model='additive', extrapolate_trend='freq',freq=24)
# Deseasonalize Soil_VWC10cm
data.Soil_VWC10cm = data.Soil_VWC10cm.values / result_add.seasonal
#Detrend ST_10cm
data.Soil_VWC10cm = data.Soil_VWC10cm.values - result_add.trend

# Additive Decomposition
result_add = seasonal_decompose(data['Soil_VWC36cm'], model='additive', extrapolate_trend='freq',freq=24)
# Deseasonalize Soil_VWC36cm
data.Soil_VWC36cm = data.Soil_VWC36cm.values / result_add.seasonal
#Detrend ST_10cm
data.Soil_VWC36cm = data.Soil_VWC36cm.values - result_add.trend

# Additive Decomposition
result_add = seasonal_decompose(data['Soil_TWC'], model='additive', extrapolate_trend='freq',freq=24)
# Deseasonalize Soil_TWC
data.Soil_TWC = data.Soil_TWC.values / result_add.seasonal
#Detrend Soil_TWC
data.Soil_TWC = data.Soil_TWC.values - result_add.trend

#End of Deseasonalize and Detrend of each variable
```

Figure 25: Removal of Seasonality and Trend From Time-Series Data

As indicated in above Figure24 and Figure25 each variable is transformed by removing trend and seasonality using addictive decomposition(*Time Series Analysis in Python – A Comprehensive Guide with Examples*; 2019).

```
[ ]  #Start of Stabilize Variance using YeoJohnson used
     DataBox=data
     DataBox['ST_10cm'], lam = yeojohnson(DataBox['ST_10cm'])
     print('Lambda: %f' % lam)

     DataBox['ST_10cm'], lam = yeojohnson(DataBox['ST_36cm'])
     print('Lambda: %f' % lam)

     DataBox['Soil_VWC10cm'], lam = yeojohnson(DataBox['Soil_VWC10cm'])
     print('Lambda: %f' % lam)

     DataBox['Soil_VWC36cm'], lam = yeojohnson(DataBox['Soil_VWC36cm'])
     print('Lambda: %f' % lam)

     DataBox['Soil_VWC79cm'], lam = yeojohnson(DataBox['Soil_VWC79cm'])
     print('Lambda: %f' % lam)

     data=DataBox
     #End of Stabilize Variance using YeoJohnson used
```

Figure 26: Stabilize Variance of Time-Series Data

As represented in above Figure26, variance of each variables is stabilized using yeo-johnson transform is used.

```
[ ] #Start of standardization using Z-score
    from sklearn import preprocessing
    std_scale = preprocessing.StandardScaler().fit(data[['ST_10cm','ST_36cm' ,'Soil_VWC10cm','Soil_VWC36cm','Soil_TWC']])
    data['ST_10cm'] = std_scale.transform(data['ST_10cm'])
    data['ST_36cm'] = std_scale.transform(data['ST_36cm'])
    data['Soil_VWC10cm'] = std_scale.transform(data['Soil_VWC10cm'])
    data['Soil_VWC36cm'] = std_scale.transform(data['Soil_VWC36cm'])
    data['Soil_TWC'] = std_scale.transform(data['Soil_TWC'])
    #End of standardization using Z-score
```

Figure 27: Standardization of Time-Series Data

Data standardization done using z-score of data points presents variables as depicted in Figure27

```
[ ] #Two new variable added into dataframe
    #Day of year(1to366)
    #Hour of Day(0to23)
    df=data
    df['Day'] = df.index.dayofyear
    df['Hour'] = df.index.hour
```

Figure 28: Adding Two Variables in Dataframe

With help of index variable two new variables are added to dataframe such as day of year variable which represents values of days from 1 to 366 and hour of day variable which depict values of hours from 0 to 23.

```
[ ] #Variable for prediction
    target_names = ['Soil_VWC10cm', 'Soil_VWC36cm']

    #Shift Day for number of Days into future prediction.
    #Here Shift Day is 1 i.e. Predict given variable 1 day into future

    shift_days = 1

    #Shift data 24 time-steps
    #Here 1*24


    shift_steps = shift_days * 24  # Number of hours.
```

Figure 29: Selection of Variables for prediction and Hours in future for prediction

Prediction of volumetric soil moisture of depths 10cm and 36cm are selected as shown in above Figure29. shift_days variable is defined, it used to select number of days prediction in future for given study it is 1 day. shift_steps variable is also defined for number hours calculation for given selected days prediction.

```
[ ] #New Dataframe for shifted data
    df_targets = df[target_names].shift(-shift_steps)
```

Figure 30: New Dataframe with shift steps

New dataframe is created with selected target variables and with shifted time steps as represented in above Figure30 and it checked as well viewed using following code as shown Figure31 below,

```
[ ] #To check New dataframe shifted correctly,
    #Hence it check by comparing the original and time-shifted data-frames
    df[target_names].head(shift_steps + 5)
```

```
[ ] #This is first 5 rows of the new time-shifted data-frame.
    #This is used to compared the last 5 rows from above original dataframe output to shifted dataframe ouput,
    df_targets.head(5)
```

```
[ ] #The new time-shifted data-frame has the same length as the original data-frame
    # but it showing at last values `NaN` because data shifted
    #hence at end dataset trying to shift data that does not exist in the original data-frame.

    df_targets.tail()
```

Figure 31: View of New Dataframe with shift steps

```
[ ] #input-signals
    x_data = df.values[0:-shift_steps]
```

```
[ ] print(type(x_data))
    print("Shape:", x_data.shape)
```

```
[ ] #target-signals
    y_data = df_targets.values[:-shift_steps]
```

```
[ ] print(type(y_data))
    print("Shape:", y_data.shape)
```

Figure 32: Input-signals and Target-signals

as shown in previous figures new dataframe is created using time-shifted with selected predictive variable is now converted into arrays as shown in above Figure32. Such as for input-signals array and for target-signals array.

```
[ ] #number of observations in the dataset
    num_data = len(x_data)
    num_data
```

```
[ ] #fraction use for making training set
    train_split = 0.9
```

```
[ ] #number of observations in the training-set
    num_train = int(train_split * num_data)
    num_train
```

```
[ ] #number of observations in the test-set
    num_test = num_data - num_train
    num_test
```

Figure 33: Defining Ratio to split dataframe into Train and Test sets

```
[ ] #Number of input-signals for the training- and test-sets
    x_train = x_data[0:num_train]
    x_test = x_data[num_train:]
    len(x_train) + len(x_test)
```

```
[ ] #Number of output-signals for the training- and test-sets
    y_train = y_data[0:num_train]
    y_test = y_data[num_train:]
    len(y_train) + len(y_test)
```

```
[ ] #number of input-signals
    num_x_signals = x_data.shape[1]
    num_x_signals
```

```
[ ] #number of output-signals
    num_y_signals = y_data.shape[1]
    num_y_signals
```

Figure 34: Train and Test sets of Input signals and Output signal

The Figure33 represents, ratio for split the dataframe into train and test set. For this research 90 : 10 ratio is selected which means 90% data of selected site will used to train LSTM and ANN models and remain 10% used for testing the LSTM and ANN models. The Figure34 denote procedure of forming input-signals and target-signals with receptive to their train and test sets.

```
[ ] #To check range of values
    print("Min:", np.min(x_train))
    print("Max:", np.max(x_train))
```

```
[ ] #creation of  a scaler-object for the input-signals
    #to scale the data before apply asinput to the neural networks,
    #Using scikit-learn
    x_scaler = MinMaxScaler()
```

```
[ ] # To find out range of values present in the training-data
    # and then training-data is scaled.
    x_train_scaled = x_scaler.fit_transform(x_train)
```

```
[ ] # the scaled data is now between 0 and 1.
    print("Min:", np.min(x_train_scaled))
    print("Max:", np.max(x_train_scaled))
```

Figure 35: Scaling of Train and Test Sets

```
[ ] #Scaler object used as well for to scale the input-signals in the test-set.
    x_test_scaled = x_scaler.transform(x_test)
```

```
[ ] #New scaler-object for the target-data.
    y_scaler = MinMaxScaler()
    y_train_scaled = y_scaler.fit_transform(y_train)
    y_test_scaled = y_scaler.transform(y_test)
```

```
[ ] #shapes of the input Data
    print(x_train_scaled.shape)
    #shapes of the output data
    print(y_train_scaled.shape)
```

Figure 36: Scaling of Train and Test Sets

The neural network works best when input and target signals are in range of -1 to 1. Hence using sckit-learn's MinMaxScaler used to scaled it. In Figure35 MinMaxScaler object is created for Input-signals and in Figure36 MinMaxScaler object is created for Target-signals as well.

```
[ ] #function to create a batch of shorter sub-sequences from at random from the training-data

    def batch_generator(batch_size, sequence_length):
        """
        Generator function for creating random batches of training-data.
        """

        # Infinite loop.
        while True:
            # Allocate a new array for the batch of input-signals.
            x_shape = (batch_size, sequence_length, num_x_signals)
            x_batch = np.zeros(shape=x_shape, dtype=np.float16)

            # Allocate a new array for the batch of output-signals.
            y_shape = (batch_size, sequence_length, num_y_signals)
            y_batch = np.zeros(shape=y_shape, dtype=np.float16)

            # Fill the batch with random sequences of data.
            for i in range(batch_size):
                # Get a random start-index.
                # This points somewhere into the training-data.
                idx = np.random.randint(num_train - sequence_length)

                # Copy the sequences of data starting at this index.
                x_batch[i] = x_train_scaled[idx:idx+sequence_length]
                y_batch[i] = y_train_scaled[idx:idx+sequence_length]

            yield (x_batch, y_batch)
```

Figure 37: Batch Generator Function

```
[ ] #batch-size
    batch_size = 256


    #Random sequence is of 8 weeks is selected for prediction process,
    #one time step is equal to 1 hour
    #hence 24*7 is equal to week
    #and 24*7*8 is equal to 8 week.

    sequence_length = 24 * 7 * 8
    sequence_length
```

```
[ ] #creation of batch-generator.
    generator = batch_generator(batch_size=batch_size,sequence_length=sequence_length)
```

Figure 38: Calling Batch Generator Function with defined parameters

The Figure37 denote definition of batch generator function and Figure38 depict calling user-defined batch generator function using selected parameters. The purpose of this function is instead of training of whole models on complete train set at once instead it creates batch of training dataset. Basically, this function creates two arrays, first

represents batch of input signals and second array represents batch of output signals. These two array are filled using by taking random index for the training set and then data is copied to each of this arrays by data which starting at this index. While calling this function two parameters are passed, first parameter is for batch size and which is 256 to make use of full GPU strength for given task of prediction. Second parameter is for sequence length, for this research 8-week time-steps is selected which means each random sequence gives 8-weeks observations. Total sequence length is formed like this, one time-step formed by using one hour which means in one day total 24 observations. To count it for one week, it is like 24×7. And to covered given observation periods i.e. 8-weeks, is calculated like this 24×7×8 which gives 1344. So, sequence length and batch size for above Figure38 function is 1344 and 256 respectively. This function is for LSTM as well for ANN model to train on each site.

```
[ ] # to check batch-generator.
    x_batch, y_batch = next(generator)

    #Batch of input-signals and output-signals.
    print(x_batch.shape)
    print(y_batch.shape)
```

```
[ ] #Validation Set
    validation_data = (np.expand_dims(x_test_scaled, axis=0),
                       np.expand_dims(y_test_scaled, axis=0))
```

Figure 39: Validation Sets

To avoid problem that model performed well on training set so that it does not generalize well on testing data i.e. problem of overfitting. By keeping this point in mind hence performance of each models is observed for after each epoch on test set. If the performance of model seen improving on test set then only model weights are stored. As compared to batches for training the purposed model using batch generator function but for testing whole sequence of data is passed from given test set and then after prediction accuracy measured on that whole testing set.

```
#Long Short Term Memory (LSTM) to the network. This will have 512 outputs for each time-step in the sequence.
#batch of sequences of arbitrary length indicated by 'None',
#and each observation has a number of input-signals indicated by 'num_x_signals'.

#Dropout layer is added,
#dropout rate is set to 20%, meaning one in 5 inputs will be randomly excluded from each update cycle.

#because of scaler-object output signals are limited to between 0 to 1
# Hense in last layer, output of the neural network are limited to be between 0 and 1 using 'Sigmoid' Activation Function,
#last layer is output layer and It only has two node i.e 'num_y_signals', which is for prediction.

model.add(LSTM(units=512, return_sequences=True, input_shape=(None, num_x_signals,)))
model.add(Dropout(0.2))
model.add(Dense(num_y_signals, activation='sigmoid'))
```

Figure 40: Layers for LSTM Model

Above Figure40 denote LSTM model. As shown in first layer, for this model 512 is selected as units it nothing but for each time-steps in the sequence it will have 512 outputs. batch of sequences of arbitrary length indicated by 'None' and each observation has a number of input-signals indicated by 'num_x_signals'. Second layer is dropout layer, dropout rate set to 20% which means one in 5 inputs will be randomly excluded from each update cycle. Third layer is dense layer, this layer is nothing but output layer for LSTM model and using 'Sigmoid' as activation function it limits output of network to 0 to 1 . This output layer has only only two node i.e. 'num_y_signals' which represents for prediction.

```
[ ] #use a linear activation function on the output instead. This allows for the output to take on arbitrary values.
    #It might work with the standard initialization for a simple network architecture, but for more complicated network architectures

    if False:
        from tensorflow.python.keras.initializers import RandomUniform

        # Maybe use lower init-ranges.
        init = RandomUniform(minval=-0.05, maxval=0.05)

        model.add(Dense(num_y_signals,
                        activation='linear',
                        kernel_initializer=init))
```

```
[ ] warmup_steps = 50
```

Figure 41: Additional Dense Layer with Linear Activation Function

```
[ ] def loss_mse_warmup(y_true, y_pred):
        """
        Calculate the Mean Squared Error between y_true and y_pred,
        but ignore the beginning "warmup" part of the sequences.

        y_true is the desired output.
        y_pred is the model's output.
        """

        # The shape of both input tensors are:
        # [batch_size, sequence_length, num_y_signals].

        # Ignore the "warmup" parts of the sequences
        # by taking slices of the tensors.
        y_true_slice = y_true[:, warmup_steps:, :]
        y_pred_slice = y_pred[:, warmup_steps:, :]

        # These sliced tensors both have this shape:
        # [batch_size, sequence_length - warmup_steps, num_y_signals]

        # Calculate the MSE loss for each value in these tensors.
        # This outputs a 3-rank tensor of the same shape.
        loss = tf.losses.mean_squared_error(labels=y_true_slice,
                                            predictions=y_pred_slice)

        # Keras may reduce this across the first axis (the batch)
        # but the semantics are unclear, so to be sure we use
        # the loss across the entire tensor, we reduce it to a
        # single scalar with the mean function.
        loss_mean = tf.reduce_mean(loss)

        return loss_mean
```

Figure 42: Mean Squared Error loss Function

To check performance of LSTM model that is how well model performed in prediction of values and how close prediction values to true values is done using above user-defined function shown in Figure42. For this proposed study Mean Squared Error(MSE) is used to check loss in prediction process. As proposed models are using input signal for few time step at very beginning. so, it may possible that output is inaccurate. Hence then models would try to use loss-value in at early time-steps to get accurate output but this may twist out shape of model. By keep this thing in mind proposed model made by giving a warmup-period of 50 time steps to get accuracy in output in later time-steps.

```
[ ] #optimizer and learning-rate
    optimizer = RMSprop(lr=1e-3)
```

```
[ ] #compile the model for training
    model.compile(loss=loss_mse_warmup, optimizer=optimizer)
```

```
[ ] #Model Summary of LSTM Model
    model.summary()
```

Figure 43: Compile and Summary of LSTM Model

Above Figure43 depict LSTM model comiple and summary code. As shown above $RMSprop$ is used as optimizer for given model with learning rate of $1e-3$. It also shows that previously mentioned loss_mse_warmup is used as loss. LSTM model summary also find out using above mentioned code.

```
[ ] #Callback Functions
    path_checkpoint = '23_checkpoint.keras'
    callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint,
                                          monitor='val_loss',
                                          verbose=1,
                                          save_weights_only=True,
                                          save_best_only=True)


    callback_early_stopping = EarlyStopping(monitor='val_loss',
                                            patience=5, verbose=1)


    callback_tensorboard = TensorBoard(log_dir='./23_logs/',
                                       histogram_freq=0,
                                       write_graph=False)

    callback_reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                                           factor=0.1,
                                           min_lr=1e-4,
                                           patience=0,
                                           verbose=1)

    callbacks = [callback_early_stopping,
                 callback_checkpoint,
                 callback_tensorboard,
                 callback_reduce_lr]
```

Figure 44: Definition of Callback Functions

As denoted in Figure44 four callback functions used in developing LSTM model. While training the proposed models all callbacks i.e. values of checkpoints are saved for keras using various callback functions. In this study total four callback function are implemented for various proposed use. First callback used to save all checkpoint during training of model. Second callback used to stop the optimization of model when performance of proposed model getting exacerbate on given validation set. Third callback function used for saving TensorBoard log while training the model. Fourth callback function used to monitor validation loss, basically it used to reduce given learning rate for optimizer function when loss of validation is not improving since last epoch. fourth callback function contains factor value 0.1 and when this function needed to reduce learning rate then it multiply by factor value and learning rate don't get reduce below the 1e 4 as defined in fourth callback function.

```
[ ] #Train the LSTM
    %%time
    model.fit_generator(generator=generator,
                        epochs=20,
                        steps_per_epoch=100,
                        validation_data=validation_data,
                        callbacks=callbacks)
```

Figure 45: Train The LSTM Model

Whole proposed models are implemented in google cloud service with 13Gb NVIDIA's Tesla K80 GPU. As shown in above Figure45 training of LSTM. As shown model is trained with 20 epochs and each epochs consist of 100 steps in each epoch. It also shows parameters such as generator, validation data and callback functions.

```
[ ] #Load Checkpoint
    try:
        model.load_weights(path_checkpoint)
    except Exception as error:
        print("Error trying to load checkpoint.")
        print(error)
```

Figure 46: Load-Check points after LSTM Model Training

It may possible LSTM model performance had worse on testing set for several epochs just before training of LSTM model stops. Therefore, to get last saved checkpoint which had the best performance on the testing set and it done by using as shown above in Figure46.

```
[ ] #Performance on Test-Set
    result = model.evaluate(x=np.expand_dims(x_test_scaled, axis=0),
                            y=np.expand_dims(y_test_scaled, axis=0))
```

```
[ ] print("loss (test-set):", result)
```

Figure 47: Result of LSTM Model Evaluation on Test-set

The Above represented function in Figure47 is for to evaluate the LSTM model's performance on the test-set.

```
[ ]  #Function to genrate prediction

     def plot_comparison(start_idx, length=100, train=True):
         """
         Plot the predicted and true output-signals.

         :param start_idx: Start-index for the time-series.
         :param length: Sequence-length to process and plot.
         :param train: Boolean whether to use training- or test-set.
         """

         if train:
             # Use training-data.
             x = x_train_scaled
             y_true = y_train
         else:
             # Use test-data.
             x = x_test_scaled
             y_true = y_test

         # End-index for the sequences.
         end_idx = start_idx + length
```

Figure 48: Function to plot the Predicted and True Output-Signals

```
         # Select the sequences from the given start-index and
         # of the given length.
         x = x[start_idx:end_idx]
         y_true = y_true[start_idx:end_idx]

         # Input-signals for the model.
         x = np.expand_dims(x, axis=0)

         # Use the model to predict the output-signals.
         y_pred = model.predict(x)

         # The output of the model is between 0 and 1.
         # Do an inverse map to get it back to the scale
         # of the original data-set.
         y_pred_rescaled = y_scaler.inverse_transform(y_pred[0])

         # For each output-signal.
         for signal in range(len(target_names)):
             # Get the output-signal predicted by the model.
             signal_pred = y_pred_rescaled[:, signal]

             # Get the true output-signal from the data-set.
             signal_true = y_true[:, signal]

             # Make the plotting-canvas bigger.
             plt.figure(figsize=(15,5))
```

Figure 49: Function to plot the Predicted and True Output-Signals

```
        # Plot and compare the two signals.
        plt.plot(signal_true, label='true')
        plt.plot(signal_pred, label='pred')

        # Plot grey box for warmup-period.
        p = plt.axvspan(0, warmup_steps, facecolor='black', alpha=0.15)

        # Plot labels etc.
        plt.ylabel(target_names[signal])
        plt.legend()
        plt.show()
```

Figure 50: Function to plot the Predicted and True Output-Signals

The above Figure48 to Figure50 definition of user-defined function to plot predicted values and true values in one plot. Where, start_idx represnts index for true and predicted values to start from. length represents length of data points from given start_idx. train represent a Boolean variable which show, if True then it compared true and predicted values for training set and if false then it compared true and predicted values for testing set.

```
[ ] # from the test-data.
    plot_comparison(start_idx=200, length=1000, train=False)
```

Figure 51: Plot the Predicted and True Output-Signals

A previously mentioned user-defined function to plot predicted values and true values is called as in Figure51 to plot using Testing data.

```
[ ] #Input signals for ANN
    x_test_scaled
    y_test_scaled
    x_train_scaled
    y_train_scaled
```

Figure 52: List of Input and Output Signals For ANN

The Figure52, shows list of signals used for the ANN model.

```
[ ] #First Layer of ANN having 100 nodes,
    #activation fuction is 'relu',
    #batch of sequences of arbitrary length indicated by 'None',
    #and each observation has a number of input-signals indicated by 'num_x_signals'.

    #Second Layer is hidden layer of this ANN model with 100 nodes,
    #activation function is 'relu'.

    #Third layer, activation fuction is 'relu',
    #last layer is output layer and It only has two node i.e 'num_y_signals', which is for prediction.
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Dense(100, activation=tf.nn.relu,input_shape=(None, num_x_signals,)))
    model.add(tf.keras.layers.Dense(100, activation=tf.nn.relu))
    model.add(tf.keras.layers.Dense(num_y_signals, activation=tf.nn.relu))
```

Figure 53: Layers for ANN Model

As represented in above Figure53 ANN model. First layer, contains 100 nodes with 'relu' as activation function. Batch of sequences of arbitrary length is indicated by 'None' and each observation has a number of input-signals which indicated by 'num_x_signals'. Second layer, is hidden layer with 100 nodes with activation function as 'relu'. Third layer, this layer is output layer of ANN with 'relu' as activation function and it has only two nodes i.e. 'num_y_signals' which indicates number of prediction variables.

```
[ ] #compile the model for training
    model.compile(loss="mean_squared_error", optimizer=optimizer)
```

```
[ ] #Model Summary of ANN Model
    model.summary()
```

Figure 54: Compile and Summary of ANN Model

In above Figure54 depict ANN model's compile and summary code, loss with mean squared error with optimizer mentioned in above Figure43 also used in training of ANN model.

```
[ ] #Train the ANN
    %%time
    model.fit_generator(generator=generator,
                        epochs=20,
                        steps_per_epoch=100,
                        validation_data=validation_data)
```

```
[ ] #Performance on Test-Set
    result = model.evaluate(x=np.expand_dims(x_test_scaled, axis=0),
                            y=np.expand_dims(y_test_scaled, axis=0))
```

```
[ ] print("loss (test-set):", result)
```

Figure 55: Training and Result of ANN Model

Whole proposed models are implemented in google cloud service with 13Gb NVIDIA's Tesla K80 GPU. As shown in above Figure55 training of ANN. As shown model is trained with 20 epochs and each epochs consist of 100 steps in each epoch. It also shows parameters such as generator and validation data. The above Figure also shows ANN model's performance on the test-set.

In below called function, as previously mentioned in Figure48 to Figure50 defined user-defined function is used for ANN model as well.

```
[ ] # from the test-data.
    plot_comparison(start_idx=200, length=1000, train=False)
```

Figure 56: Plot the Predicted-Signals and True Output-Signals

A previously mentioned user-defined function to plot predicted values and true values is called as shown in Figure56.

References

SINGH, A. (2018). A gentle introduction to handling a non-stationary time series in python.
URL: *https://www.analyticsvidhya.com/blog/2018/09/non-stationary-time-series-python/"*

Stern, M. A., Anderson, F. A., Flint, L. E. and Flint, A. L. (2018). Soil moisture datasets at five sites in the central sierra nevada and northern coast ranges, california, *U.S. Geological Survey Data Series* **1083**.

Time Series Analysis in Python – A Comprehensive Guide with Examples (2019).
URL: *https://www.machinelearningplus.com/time-series/time-series-analysis-python/*