Prediction of new corona cases using Neural Network with Regularization

By:

Ahmad Nadeem Saigol

Contents

Dataset:

As it was required to train a model for both US and worldwide such that they could predict the new corona cases, thus two separate datasets were necessary. The dataset has been transformed in such a way that the model will predict new cases per month. A brief review is given below:

For Worldwide:

- The dataset for worldwide was directly downloaded from 'https://ourworldindata.org/'.
- Initially the size of the dataset was (56854,52) but it had a lot of unnecessary and redundant data was removed and was further transformed to size of (342,7). The data for worldwide is dated from 21/22/2020 to 12/29/2020.
- Following features are selected for the training of the model:
    - Month
    - New Cases per million
    - New Deaths
    - New Deaths per million
- The dataset is scaled between 0 and 1.
- Column 'State' is removed from the dataset using the training code as it is for reference only.

For United States:

- The data for US was retrieved from "COVID-19 Data Repository by the Center for Systems Science and Engineering (CSSE) at Johns Hopkins University" Github link: https://github.com/CSSEGISandData/COVID-19.
- After reading all .csv files in python, the data was of size (25372,25) containing data from 04/12/2020 to 12/30/2020. It was then cleaned and transformed to size of (350, 7).
- The features which were deemed appropriate and whose data was available consisted of
    - Month
    - Latitude
    - Longitude
    - Incident Rate: cases per 100,00 persons.
    - New tests per month
- The dataset is scaled between 0 and 1.
- Columns: 'locations' and 'dates' is removed from the dataset using training code as they are for reference only.

Class Dataset:

This is the class in NN_Training_Code.ipynb' which is responsible for reading data, removing any feature not to be included, scaling the dataset, and splitting the data. Details are as follows:

- The constructor takes two arguments:
  - 'Dir': path to the dataset
  - 'remove_cols': for any column not to be added in the features. (default value= empty list)
- A method 'scale_dataset' can be called on the object of the class for scaling of the dataset. This method modifies the original dataset. Further, it allows two types of scaling which can be used by setting the value of parameter 'st':
  - True: for performing the scaling between 0 and 1
  - False: for performing the scaling between -1 and 1
- There is another function 'split_data' in this class which carries out the splitting of data into training, validation, and testing dataset. This is done by setting the corresponding argument to True (train, valid or test) and it returns the corresponding dataset. The splitting ratio is:
  - Training data: 60%
  - Validation data: 20%
  - Testing data: 20%
- Because of the way the class is implemented, these functions would have to be called on the object to carry out the transformation of the matrix.

Neural Network:

There is another class in 'NN_Training_Code.ipynb' named as 'Network'. This is the heart of the whole code. With the help of number of small functions and attributes, it carries out forward propagation, back propagation, gradient checking, and training of the neural network. However, the user is only supposed to create an object of the class and call the method 'train' on it. Except for the gradient checking, everything is taken care off. Point to note is that throughout the network, the calculations are carried using matrices. A detailed description of the network is provided below:

Constructor:

It takes four arguments named as:

- 'Hidden_units': this basically describes the network architecture. It must be provided as list with number of neurons in each layer as element of the list. For example, if the list [3,2] is passed, then the first hidden layer will have three neurons and the second hidden layer will have two neurons.
- 'Activation_fnt': The class allows two type of activations which could be applied on the hidden layers: 'Sigmoid' and 'ReLU' (same can only passed to the parameter). With the help of this parameter, the constructor sets the corresponding activation function and its derivative. In case of any other value is provided, the program will terminate with an error message. Further, each of them has their own function in the class:
  - 'SigmoidFnt': applies the sigmoid function on the input and returns the result.

$$g(z) = \frac{1}{1 + e^{-z}}$$

- ‘SigmoidDerivative’: calculates the derivative of the sigmoid function using the input and returns the result. (element-wise multiplication)

$$g'(z)=A\ (1-A)$$

- ‘ReLU’: applies ReLU function on the input and returns the result.

$$g(z)=\max\ (0,\ Z)$$

- ‘ReLUDerivative’: calculates the derivative of the ReLU function using the input and returns the result.

$$g'(z)= 0 \text{ if } A < 0$$
$$g'(z) = 1 \text{ if } A > 0$$

- ‘Train_data’: the data on which training needs to be carried out. It must be a tuple with first element as X (dim: (m, n)) and second element Y (dim: (m,1)) where m represents the number of examples and n represents the number of features. This tuple is then destructed and stored separately as attributes of the class.
- ‘Valid_data’: same as the train_data but it is the dataset on which the dataset will be validated.

This constructor also setups a dictionary which is basically used for the storing data of each layer. Each layer has its own dictionary and has keys: ‘thetas’ (not for last layer), ‘activations’ and ‘derivatives’ (not for last layer). Further it also calculates and stores the total number of layers.

Forward Propagation:

It is performed using the function ‘forward_propagate’. It calculates and stores activations of each layer in the dictionary with the help of thetas stored in the same dictionary. For the input layer, it sets activations equal to the inputs. For all other layers, firstly it adds bias terms to the activations of the previous layer using ‘AddBias’ function:

- ‘AddBias’: this function adds column of 1 as first column of the input and returns it.

Then it calculates weighted average of the activations using ‘LinearFnt’ function:

- ‘LinearFnt’: this function carries out the dot product of X and theta and returns it.

$$z^{(L+1)} = a^{(L)}\ \theta^{(L)}$$

Where L=number of layer

After that, if the layer is not output layer, activation function (described in the previous section) is applied to these z’s and is stored in the dictionary.

Back Propagation:

It is carried out using the function 'back_propagate' and calculates the partial derivative of the cost function with respect to each weight in the network. It calculates small deltas, upper deltas, and partial derivatives from the output layer to input layer. For output layer, it calculates the difference between the predicted output and actual values.

$$\delta^{(L)} = a^{(L)} - y^{(L)}$$

Then it calculates the upper delta of the previous layer using the formula ('LinearFnt' (dot product)):

$$\Delta^{(L-1)} = (a^{(L-1)})^T \delta^{(L)}$$

Then it calculates the partial derivatives w.r.t previous layer thetas using the formula('Derivatives'):

$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \text{ if } j \neq 0$$

$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} \text{ if } j = 0$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

- 'Derivatives': calculates the partial derivatives and returns it. It takes lambda as one of the inputs which is the regularization parameter. To compensate for bias term, it first creates a matrix of size of theta filled with lambda values and then it assigns zero value to the first row.

For all other layers, it first calculates the global gradient using the function 'GlobalGrad' which calls the 'LinearFnt' to calculate the dot product of theta of the layer and small delta of the next layer:

$$\text{Global Gradient} = \delta^{(L)} (\theta^{(L+1)})^T$$

Then it calculates local gradient which is basically equal to the derivative of the activation applied on the layer. After that, it performs element wise multiplication of local gradient (delta of the bias removed) and global gradient to obtain the lower delta. Then it calculates upper delta and partial derivatives using the same methods described above. All activation values and thetas are obtained from the network dictionary and all partial derivatives are stored in the same dictionary.

Gradient Checking:

To make sure the derivatives calculated using the method described above is correct, for few examples the gradient is also calculated numerically. Important point to note is that the code is generic enough to calculate the derivatives for all examples but as this function is computationally expensive, only few examples were and should be passed.

This is achieved by the 'gradient_check' function. This function has its own dictionary which stores original weights, the values of derivatives calculated using the above method, the values of

derivatives calculated numerically and the percentage difference between them. This dictionary is printed out at the end of the function for the confirmation that the derivatives calculated using the above method are correct.
For each value of theta in each layer, initially, it adds a small epsilon value to it, performs forward propagation, calculates, and stores cost. Then it subtracts the same epsilon value from original theta, performs forward propagation, calculates, and stores cost. Then it calculates the difference between the two costs and divides it by two times of epsilon.

$$d/d\theta\ (J(\theta)) = [J(\theta+\varepsilon)-J(\theta-\varepsilon)]/2\varepsilon$$

Epsilon is a matrix of zeros of size of theta of the layer in the code with epsilon value only on the index of which theta is being considered at the time. To store the derivatives calculated numerically, a matrix of zeros of size of theta of the layer is initialized and its values are replaced one by one with the numerically calculated derivatives.

Training:

This function trains the neural network using Mini-Batch Gradient Descent and is achieved through 'train' function on the class object. It takes number of arguments:

- 'batch_size': this is number of the examples after which weights are updated
- 'epochs': Number of times the complete training dataset needs to be passed through the neural network
- 'lr': Learning rate
- 'lmbda': Regularization Parameter

First, it initializes weights of the network by calling the function 'randomly_init_weights'

- 'randomly_init_weights': firstly, it determines the shape of theta of each layer in the network using the 'hidden_units'. It takes the form of (m, n) where m is the number of units in previous layer (including bias) and n is the number of units in next layer. Then it initializes each theta with values obtained from uniform distribution between 0 and 1.

Then it has two loops: one for epoch and one for batch. Inside the nested loop, it finds indexes of the batch and stores it. Then it performs forward propagation and calculates cost using the function 'MSE':

- 'MSE': this function calculates the mean square error between predicted output and actual output.

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h\left(x^{(i)}\right) - y^{(i)}\right)^2$$

This cost is stored in the list for the plotting learning curves. After that, it performs back propagation and it updates the parameters by calling the function 'update_weights':

- 'update_weights': this function updates the values of thetas using Gradient descent:

$$\theta := \theta - lr*d/d\theta\ (J(\theta))$$

Then, it performs the process of forward propagation and cost computation and storing for validation data. After training the model, it returns the cost values for both training and validation data per iterations.

Hypothesis:

The activations for each hidden layer are calculated using the formula:

$$a^{(L+1)} = g(a^{(L)}\,\theta^{(L)})$$

Where 'g' represents activation function

For first hidden layer, inputs to the network becomes the activation. In case of output layer, no activation is applied as regression problem is being solved.

Objective Function:

The objective function or cost function which is used for optimizing the thetas is as follows:

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

$$where\ m = number\ of\ training\ examples$$
$$n = number\ of\ parameters$$
$$\lambda = regularization\ parameter$$

However, for evaluation the cost functions used are:

$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$J_{cv}(\theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}\left(h_\theta\left(x_{cv}^{(i)}\right) - y_{cv}^{(i)}\right)^2$$

$$J_{test}(\theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{test}}\left(h_\theta\left(x_{test}^{(i)}\right) - y_{test}^{(i)}\right)^2$$

Model:

The model will predict the number of new cases per month using the feature set and architecture used.

For Worldwide:

- The model has 6 neurons in first layer, 4 neurons in second layer, 3 neurons in third layer and 2 neurons in fourth layer.
- The activation function used is Sigmoid Function.
- The batch size is set to 50.
- Number of epochs is set to 10
- Learning Rate is 0.01
- The Regularization parameter is equal to 0.2

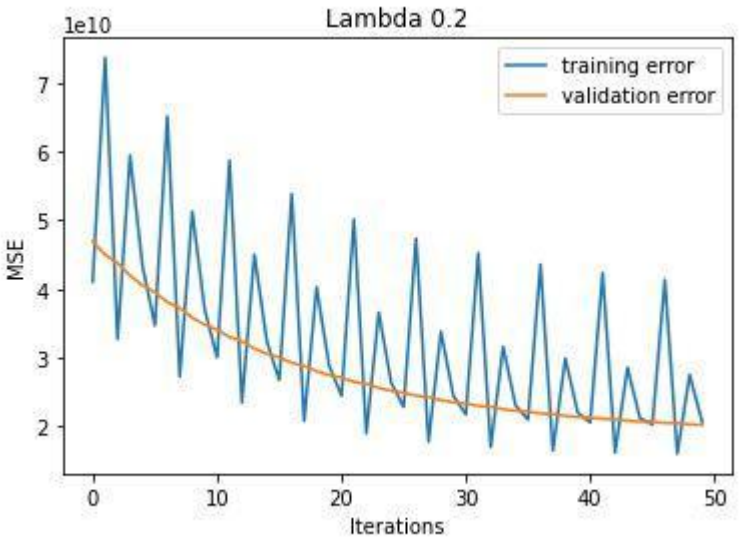
For US:

- The model has 3 neurons in first layer, and 5 neurons in second layer.
- The activation function used is Sigmoid Function.
- The batch size is set to 50.
- Number of epochs is set to 20.
- Learning Rate is 0.01.
- The Regularization parameter is equal to 0.1.

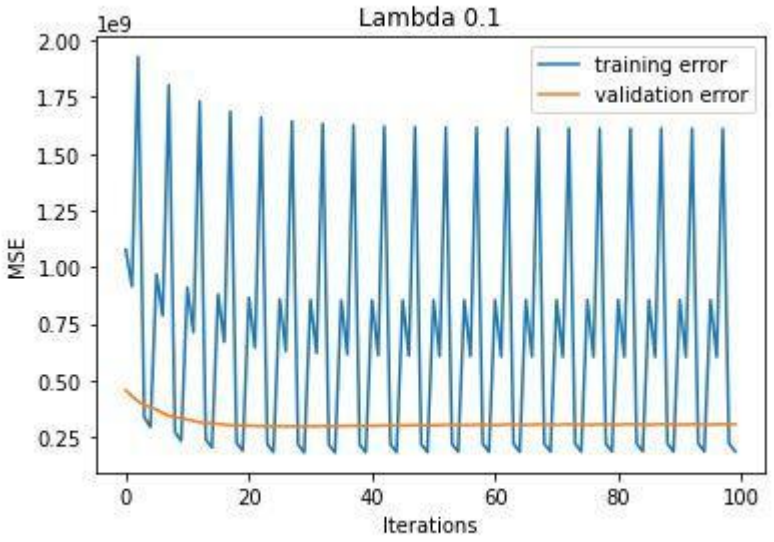
Plotting:

A plotting function is defined which takes costs as parameters and plots the graph between them. Cost is on y-axis and iterations on y-axis. This plot shows how training and validation cost changes over the time (or with each iteration).

For Worldwide:



For US:



Saving Thetas and Metadata for Future Use:

Some of the model as well as dataset properties are needed for the prediction code and thus these values are stored in special numpy format(.npy). The values saved are:

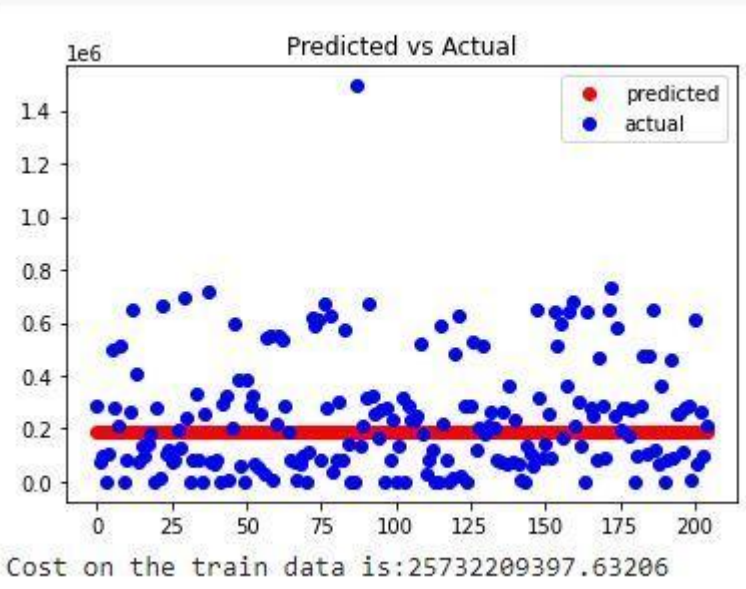
- Number of Layers
- Activation Function: ‘Sigmoid’ or ‘ReLU’
- Theta of each layer in the network
- List of column names to be removed from the dataset
- Scale: ‘True’ or ‘False’
- Scaling Type: ‘True’ ([0,1]) or ‘False’ ([-1,1])
- Minimum values of features
- Maximum values of features

Mean Error:

A function ‘prediction’ is defined for carrying out the prediction and the calculation of the Mean square error on the data. Moreover, it returns predicted outputs and cost. This function is applied on training, validation, and test dataset. For each dataset, the returned cost is printed and a scatter plot between actual values and predicted values is plotted.

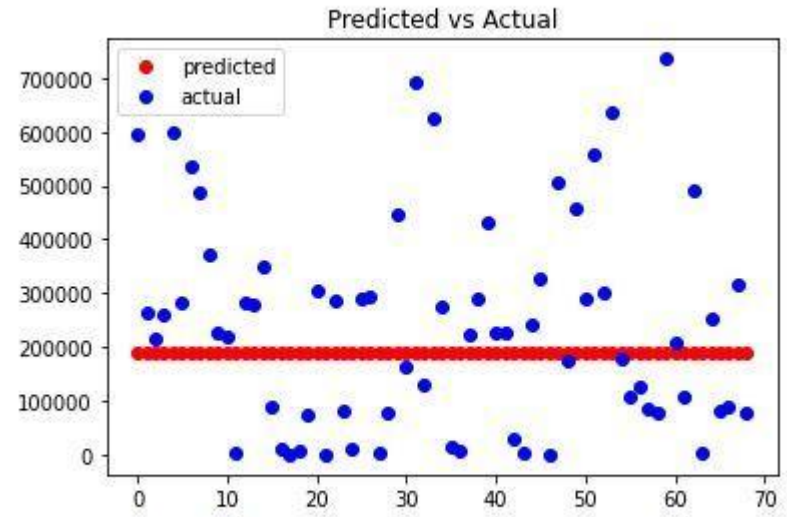
For Worldwide:

Training Dataset:



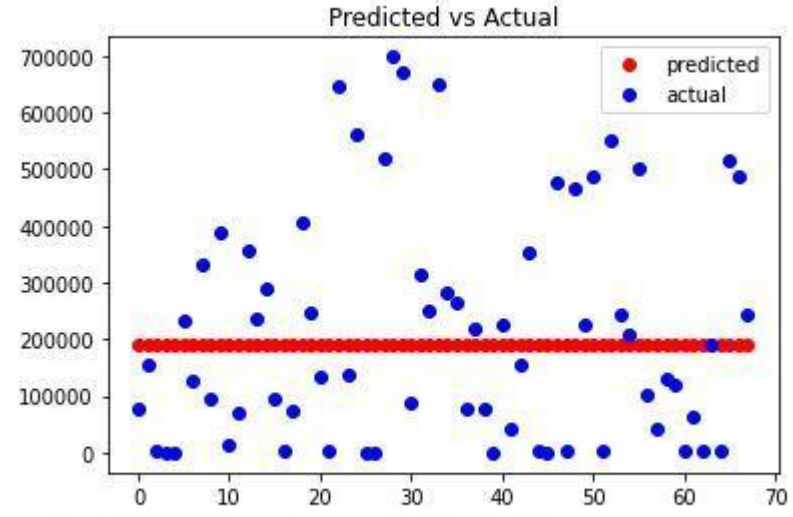
Cost on the train data is:25732209397.63206

Validation Dataset:




```
Cost on the validation data is:20134309996.596268
```

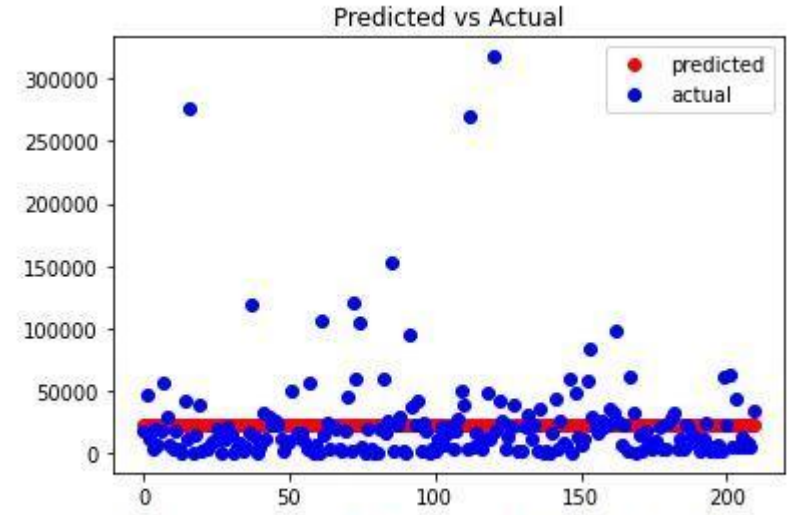
Test Dataset:




```
Cost on the test data is:20433292389.27617
```

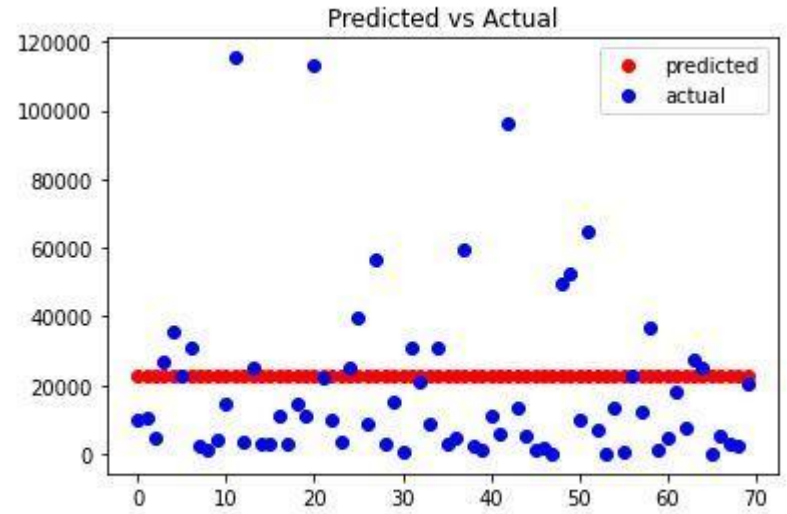
For US:

Training Dataset:



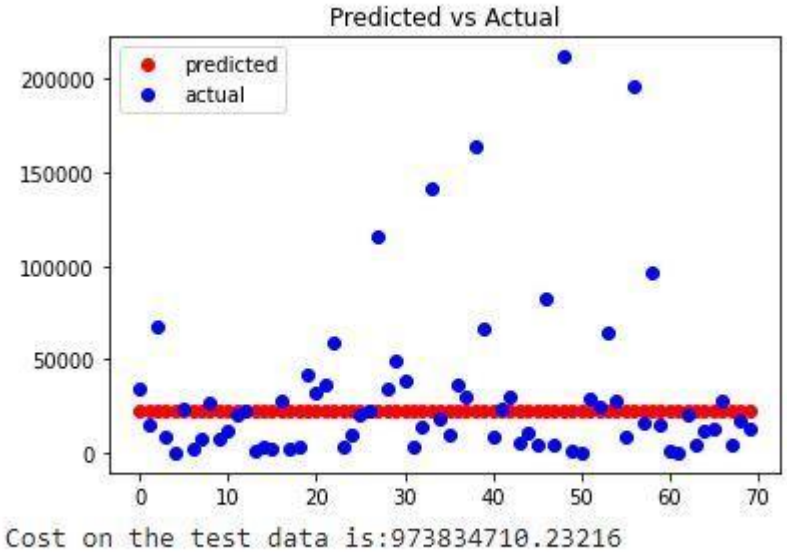
Cost on the training data is:792779546.2609148

Validation Dataset:



Cost on the validation data is:306393299.9234915

Test Dataset:



Prediction Code:

To make predictions on the unknown dataset, a separate file 'NN_Prediction_Code.ipynb' is provided. This separates the training code and makes it easier for the user to estimate the outputs on his dataset. This file has a class 'Prediction' which takes two arguments:

- 'dir_to_data' : this refers to the path of user's dataset
- 'dir_to_thetas_metadata': this refers to the path of the file(.npy) created while training the model.

It reads the user's data from his file and drops the rows with insufficient data. Then it opens the file of thetas_metadata and starts reading data one by one in the same order in which it was stored in the training code. Further, as it reads the data, it performs the number of operations such as set activation function, remove columns from the dataset etc. Moreover, it also checks whether the dataset provided by the user has correct number of features or not. Apart from the number of small functions (same as in training code), it has a function 'prediction' which carries out scaling of data (if any), performs the forward propagation, displays and returns the predicted values. More details on how to run this code is provided in Annex A.

Annexes:

Note: The Code was written using Google Colab and it is thus formatted in that way. Incase you want to re-run using other editor/environment, you would have to adjust the code accordingly to see the outputs or some outputs will not be visible because of the way Jupyter Notebook works. Further, although the complete codes are provided in the annexes, but it is recommended to read the code through .ipynb file.

Annex A: Instructions on running the code

- Open the file ‘NN_Prediction_code.ipynb’ in a python environment and place the file named as ‘Optimal_thetas_of_Neural_Networks_with_metadata_*.npy’ in the same directory as that of .ipynb.
- Create a .csv file in the same directory. In this file, add each feature name and reference name in a row in the order described by in ‘Look_up_table’ or template file. Add values under each column. Some of these values can be found in the Look_up_table file while others need to be supplied by you.
- A template file for both (US and worldwide) is provided for reference. You can use simply edit these files and use them.
- Each row represents single example and thus you can make multiple predictions at the same time by filling multiple rows.
- In the .ipynb file, run the libraries block and class prediction block. Then create an object of class and pass the following arguments:

    - ‘dir_to_data’ : path to your created file
    - ‘dir_to_thetas_metadata’ : path to ‘Optimal_thetas_of_Neural_Networks_with_metadata_*.npy’

  After that, call the function ‘prediction’ on the object of class and it will output the result. A template for both (worldwide and US) is available in .ipynb file. You can just simply change the directories according to your own requirement.
- Make sure to input the data correctly in your file and pass the correct directories to the code.
- If there are any missing values in your input data, the code will ignore that row and will not calculate its result.
- If the number or names of the columns are not correctly given, the code will not run.

Annex B: Training Code along with the Optimal Parameters:

Training Code:

```
import pandas as pd
import numpy as np
import sys
from matplotlib import pyplot as plt
%matplotlib inline
class Dataset:

  #constructor
  def __init__(self,dir,remove_cols=[]):

    # args:
```

```
    # dir = (str)directory to dataset with last column as target values
    # remove_cols = (list) name of columns which are not to be added in the
dataset as list

    #read data file
    self.data=pd.read_csv(dir)

    #remove columns if not required in the features
    self.remove_cols = remove_cols # for saving in the file as metadata
    if remove_cols:
      self.data=self.data.drop(remove_cols, axis=1)

    #select all feature names (except target values)
    self.features= self.data.columns[0:-1]

    #store the target name
    self.target=self.data.columns[-1]

    #Variables used while scaling the dataset
    #default values have been provided as these will be saved as metadata in
 the file
    self.scale=False
    self.scale_type=None
    self.min=None
    self.max=None


  def scale_dataset(self,st):

    #this function carries out scaling of the features
    #it can either be between 0 and 1 or between -1 and 1
    #set st=True for [0,1] and st=False for [-1,1]

    self.scale=True
    self.scale_type=st

    self.min=self.data[self.features].min()
    self.max=self.data[self.features].max()

    if st:
      self.data[self.features] = self.data[self.features].apply(lambda x: (x
-self.min)/(self.max-self.min), axis=1)
    else:
      self.data[self.features] = self.data[self.features].apply(lambda x: 2*
((x-self.min)/(self.max-self.min))-1, axis=1)


  def split_data(self,train=False, test=False, valid=False):

    # this fucntion carries out the splitting of data into train, test, and
validation datasets
    # depending upon which arg is set True.
    # Splitting ratio: 60% for  train, 20% for test, 20% for validation
    # Only one arg should be set true when calling this function
    # otherwise the dataset of the arg which is in first in arg list will be
 returned.
```

```
    #random_state set for reproducibility
    train_data, test_data, valid_data = np.split(self.data.sample(frac=1, ra
ndom_state=42), [int(0.6*len(self.data)), int(0.8*len(self.data))])

    if train:
      return (train_data[self.features].to_numpy(), train_data[self.target].
to_numpy().reshape(-1,1))
    if test:
      return (test_data[self.features].to_numpy(), test_data[self.target].to
_numpy().reshape(-1,1))
    if valid:
      return (valid_data[self.features].to_numpy(), valid_data[self.target].
to_numpy().reshape(-1,1))


#path to dataset
dir_world ='/content/World_Data_Transformed.csv'

#                                  remove these features from the dataset
dataset_world = Dataset(dir=dir_world, remove_cols=['location', 'dates'])

#print first five entries of dataset
print(dataset_world.data.head())

#Scale dataset
dataset_world.scale_dataset(st=True)
#dataset_world.scale_dataset(st=False)


#Training Data
train_data_world = dataset_world.split_data(train=True)
print(f'Shape of Training data : X = {train_data_world[0].shape} Y = {train_
data_world[1].shape}')

#Validation Data
valid_data_world = dataset_world.split_data(valid=True)
print(f'Shape of Validation data : X = {valid_data_world[0].shape} Y = {vali
d_data_world[1].shape}')

#Testing Data
test_data_world = dataset_world.split_data(test=True)
print(f'Shape of Testing data : X = {test_data_world[0].shape} Y = {test_dat
a_world[1].shape}')

#path to dataset
dir_US = '/content/US_Data_Transformed.csv'

#                               remove these features from the dataset
dataset_US=Dataset(dir=dir_US, remove_cols=['State'])

#print first five entries of dataset
print(dataset_US.data.head())

#Scale dataset
dataset_US.scale_dataset(st=True)
#dataset_world.scale_dataset(st=False)
```

```
#Training Data
train_data_US=dataset_US.split_data(train=True)
print(f'Shape of Training data : X = {train_data_US[0].shape} Y = {train_dat
a_US[1].shape}')

#Validation Data
valid_data_US=dataset_US.split_data(valid=True)
print(f'Shape of Validation data : X = {valid_data_US[0].shape} Y = {valid_d
ata_US[1].shape}')

#Testing Data
test_data_US=dataset_US.split_data(test=True)
print(f'Shape of Testing data : X = {test_data_US[0].shape} Y = {test_data_U
S[1].shape}')

class Network:

  #constructor
  def __init__(self, hidden_units, activation_fnt, train_data, valid_data):

    #args:
    #     hidden units: (list) it include both number of hidden layers and n
umber of units in each hidden layer as list
    # e.g. for a network with two hidden layers, first hidden layer has 3 un
its and second hidden layer has 2 units
    # hidden_units=[3, 2]
    #
    #     activation_fnt: (str) which will be applied to the hidden layers
    # its value can either be 'Sigmoid' or 'ReLU'
    #
    #     train_data: (tuple) containing both X and Y as tuple e.g. (X_train
, Y_train)
    #     valid_data: (tuple) containing both X and Y as tuple e.g. (X_valid
, Y_valid)

    #dictionary for storing layers of the network
    self.network={}

    #destruct training data tuple to X, Y
    self.train_X=train_data[0]
    self.train_Y=train_data[1]

    #destruct validation data tuple to X, Y
    self.valid_X=valid_data[0]
    self.valid_Y=valid_data[1]

    self.hidden_units=hidden_units
    self.no_of_layers = len(hidden_units) + 2


    self.activation_fnt=activation_fnt #for saving in the file

    #Select appropriate activation function and its derivative
    if activation_fnt=='Sigmoid':
      self.ActivationFnt=self.SigmoidFnt
      self.LocalGrad=self.SigmoidDerivative
```

```
    elif activation_fnt=='ReLU':
      self.ActivationFnt=self.ReLU
      self.LocalGrad=self.ReLUDerivative

    else:
      sys.exit(f'Please specify activation function again. It can either be
Sigmoid or RelU (in single quotes)') #exit if no


  #Add bias to the activations as column vector
  def AddBias(self, X):
    return np.c_[ np.ones(X.shape[0]), X]


  #Apply linear function
  def LinearFnt(self, X, theta):
    return np.dot(X, theta)


  #Apply Sigmoid function
  def SigmoidFnt(self, Z):
    return 1/(1+(np.exp(-Z)))


  #Calculate Derivative of Sigmoid function
  def SigmoidDerivative (self, A):
    return np.multiply(A, (1-A))


  #Apply ReLU
  def ReLU(self, Z):
    return np.maximum(0, Z)


  #Calculate Derivative of ReLU function
  def ReLUDerivative (self, A):
    return 1 * (A > 0)


  #Calculate Global Gradient
  def GlobalGrad(self, lower_delta, theta):
    return self.LinearFnt(lower_delta, np.transpose(theta))


  #Calculate Partial Derivatives
  def Derivatives(self, m, upper_delta, lmbda, theta): # m = no of examples
    lmbda = np.full(theta.shape ,fill_value = lmbda) # regularization parame
ter
    lmbda[0]=0 #No Regularization term for bias
    return ((upper_delta/m)+(np.multiply((lmbda/m), theta)))


  #Calculate Mean Squared Error (cost function)
  def MSE(self, y_pred, y):
    #args:
    #     y_pred = predicated output
    #     y = actual output
```

```
    return ((np.sum(np.square(y_pred - y)))/(2*y_pred.shape[0]))


  #Update the Thetas(weights) of each layer
  def update_weights(self, lr): #lr=learning rate
    for i in range(1, self.no_of_layers, 1):
      layer=f'layer{i}'
      self.network[layer]['thetas']=self.network[layer]['thetas']-(lr*self.n
etwork[layer]['derivatives'])


  #Randomly initialize weights for each layer with dimensions: m x n
  # m = no of units in previous layer (including bias)
  # n = no of units in next layer
  def randomly_init_weights (self):

    #determine the shape of the theta
    for i in range(1, self.no_of_layers, 1):

      if i==1: #input layer to first hidden layer
        theta_shape=(self.train_X.shape[1]+1, self.hidden_units[0])

      elif i == self.no_of_layers - 1: #last hidden layer to output layer
        theta_shape = (self.hidden_units[i-2]+1, self.train_Y.shape[1])

      else: #hidden layer to hidden layer
        theta_shape = (self.hidden_units[i-2]+1, self.hidden_units[i-1])

      #dictionary for storing data associated with each layer
      self.network[f'layer{i}']={}

      #randomly initialize weights in range [0, 1]
      self.network[f'layer{i}']['thetas']= np.random.uniform(0, 1, size=thet
a_shape)#*2*e)-e



  #Perform Forward Propagation
  def forward_propagate(self, X):
    #args:
    #     X = inputs
    #         (numpy array of dim m X n)
    #         m = no of examples
    #         n = no of features

    #starting from input layer to output layer (included)
    for i in range(1, self.no_of_layers+1, 1):

      layer = f'layer{i}' #current layer
      prev_layer=f'layer{i-1}' #previous layer

      #set activations of first layer equal to inputs
      if i==1:
        self.network[layer]['activations'] = X

      else:
```

```
        #Add bias term to the activations
        self.network[prev_layer]['activations']=self.AddBias(self.network[pr
ev_layer]['activations'])

        #Calculate z's
        z=self.LinearFnt(self.network[prev_layer]['activations'], self.netwo
rk[prev_layer]['thetas'])

        #Calculate a's by applying activation function except for last layer
 (output layer)
        if i==self.no_of_layers:
          self.network[layer]={} #for output layer
          self.network[layer]['pred_outputs'] = z # no activation function

        else:
          self.network[layer]['activations'] = self.ActivationFnt(z)


  #Perform Back Propagation
  def back_propagate(self, Y_actual, lmbda):

      #starting from output layer to input layer (not included)
      for i in  range(self.no_of_layers, 1, -1):

        layer = f'layer{i}' #current layer
        prev_layer=f'layer{i-1}' #previous layer

        if i==self.no_of_layers: #for output layer

          #calculate lower delta of this layer
          lower_delta = self.network[layer]['pred_outputs'] - Y_actual

          #calculate upper delta of previous layer
          upper_delta = self.LinearFnt(np.transpose(self.network[prev_layer]
['activations']), lower_delta) #of second last layer

          #calculate partial derivatives of previous layer
          self.network[prev_layer]['derivatives']=self.Derivatives(Y_actual.
shape[0], upper_delta, lmbda, self.network[prev_layer]['thetas'] )


        else: #for all other layers

          #calculate global gradient of this layer using lower delta of next
 layer
          global_Grad = self.GlobalGrad(lower_delta, self.network[layer]['th
etas'])

          #calculate local gradient of this layer using activations of this
layer
          local_Grad = self.LocalGrad(self.network[layer]['activations'] )

          #calculate lower delta of this layer
          lower_delta = np.multiply(global_Grad, local_Grad)[:, 1:] #removin
g the deltas of the bias terms
```

```
          #calculate upper delta of previous layer
          upper_delta = self.LinearFnt(np.transpose(self.network[prev_layer]
['activations']), lower_delta)

          #calculate partial derivatives of previous layer
          self.network[prev_layer]['derivatives']=self.Derivatives(Y_actual.
shape[0], upper_delta, lmbda, self.network[prev_layer]['thetas'] )



  #Perform Numerical Estimation of gradients
  def gradient_check(self, X, Y):

    #dictionary for storing thetas and derivatives (original, calculated, di
fference percentage) of each layer
    grad_network={}

    output_layer = f'layer{self.no_of_layers}'

    #for each layer in the original network
    for layer in self.network:

      # no derivative is associated with output layer
      if layer != output_layer:

        grad_network[layer]={}

        # reading original thetas
        grad_network[layer]['thetas_ori'] = self.network[layer]['thetas']

        #reading original derivatives for comparsion
        grad_network[layer]['deriv_ori'] = self.network[layer]['derivatives'
]

        size = grad_network[layer]['thetas_ori'].shape

        #for storing newly calcuated derviative for each weight
        deriv_cal = np.zeros(size)

        #for each theta in this layer of the network
        for i in range(0, size[0] ,1):
          for j in range(0, size[1] ,1):

            #small value which is added to each theta
            epsilon = np.zeros(size)
            epsilon[i,j] = 0.00001

            cost=[] #for storing the newly calculated cost

            #in first iteration add the epsilon value and calculate cost
            #in second iteartion subtract the epsilon value and calculate co
st
            for k in range (0, 2, 1):

              #replace theta values in the orignal network with new theta va
lues
              if k==0:
```

```
                self.network[layer]['thetas']=grad_network[layer]['thetas_or
i'] + epsilon

              else:
                self.network[layer]['thetas']=grad_network[layer]['thetas_or
i'] - epsilon

              #calculate new predicted outputs
              self.forward_propagate(X)

              #calculate and store cost for theta+e and theta-e
              cost.append(self.MSE(self.network[output_layer]['pred_outputs'
], Y))

            #calculate derivative and store it to the corresponding index
            deriv_cal[i,j]=(cost[0]-cost[1])/(2*e)

        #store and calculate percentage difference between original derviati
ve and numerically computed derivatives
        grad_network[layer]['deriv_cal']=deriv_cal
        grad_network[layer]['deriv_diff_percentage']=np.abs(grad_network[lay
er]['deriv_cal'] - grad_network[layer]['deriv_ori'])*100

    #print the network for verification of derivative calculation
    for p in grad_network:
      print(f'{p}\n')
      for o in grad_network[p]:
        print(f'{o}:\n {grad_network[p][o]}\n')



  #train the neural network
  def train(self, batch_size, epochs, lr, lmbda):

    #args:
    #      batch_size = (int) no of examples after which weights will be upd
ated
    #      epochs = (int) no of times complete dataset needs to be passed th
rough the neural network during training
    #      lr = (float/int) learning rate
    #      lmbda = (float/int) regularization parameter

    output_layer=f'layer{self.no_of_layers}'

    #temporary lists for storing the cost values in each iteration
    train_cost_iter=[]
    valid_cost_iter=[]

    self.randomly_init_weights()

    for epoch in range(epochs):

      for b in range (0, self.train_X.shape[0], batch_size):

        #For Training

        batch=slice(b, b + batch_size)
```

```
        self.forward_propagate(self.train_X[batch])

        #calculate and store training cost
        train_cost_iter.append(self.MSE(self.network[output_layer]['pred_out
puts'], self.train_Y[batch]))

        self.back_propagate(self.train_Y[batch], lmbda)

        self.update_weights(lr)

        #For Validation

        self.forward_propagate(self.valid_X)

        #calculate and store validation cost
        valid_cost_iter.append(self.MSE(self.network[output_layer]['pred_out
puts'], self.valid_Y))

    return train_cost_iter, valid_cost_iter


hidden_units_world = [6,4,3,2]
activation_function_world = 'Sigmoid'
#activation_function_world = 'ReLU'

model_world = Network(hidden_units_world,activation_function_world, train_da
ta_world, valid_data_world)

hidden_units_US = [3,5]
activation_function_US = 'Sigmoid'
#activation_function_US = 'ReLU'

model_US = Network(hidden_units_US,activation_function_US, train_data_US, va
lid_data_US)
#setting up hyperparameters
batch_size_world = 50
epochs_world = 10
lr_world = 0.01
lmbda_world = 0.2

train_cost_iter_world, valid_cost_iter_world = model_world.train(batch_size_
world, epochs_world, lr_world, lmbda_world )

#setting up hyperparameters
batch_size_US = 50
epochs_US = 20
lr_US = 0.01
lmbda_US = 0.1

train_cost_iter_US, valid_cost_iter_US = model_US.train(batch_size_US, epoch
s_US, lr_US, lmbda_US )

#plot the graph of how training and validation cost changes with each iterat
ion
def plot (train_cost, valid_cost, lmbda):
```

```
  plt.plot(train_cost,  label='training error')
  plt.plot(valid_cost, label='validation error')
  plt.title(f'Lambda {lmbda}')
  plt.xlabel('Iterations')
  plt.ylabel('MSE')
  plt.legend()
  plt.axis()
  plt.show()

plot(train_cost_iter_world, valid_cost_iter_world, lmbda_world)

plot(train_cost_iter_US, valid_cost_iter_US, lmbda_US)

print('The Optimal Thetas for the Worldwide Dataset are:')

thetas=f'thetas'

for layer in model_world.network:

    if layer != f'layer{model_world.no_of_layers}':

      print(f'{layer}')
      print(f'{model_world.network[layer][thetas]}')

print('The Optimal Thetas for the US Dataset are:')

for layer in model_US.network:

    if layer != f'layer{model_US.no_of_layers}':

      print(f'{layer}')
      print(f'{model_US.network[layer][thetas]}')

# Save Model properties to special numpy file with extension (.npy)
with open('/content/Optimal_thetas_of Neural_Network_with_Metadata_Worldwide
.npy', 'wb') as file:

  np.save(file, model_world.no_of_layers) # save number of layers

  np.save(file, model_world.activation_fnt) # save activation function

  for layer in model_world.network:

    if layer != f'layer{model_world.no_of_layers}':

      np.save(file, model_world.network[layer][thetas]) #save thetas of each
 layer

#Append Data properties to the same numpy file
with open('/content/Optimal_thetas_of Neural_Network_with_Metadata_Worldwide
.npy', 'ab') as file:

    np.save(file, dataset_world.remove_cols) #save column names to be remove
d from the features
```

```
    np.save(file, dataset_world.scale) # Whether scaling is done on the data
set or not

    np.save(file, dataset_world.scale_type) #If scaling is done, which one i
s carried out

    np.save(file, dataset_world.min) #min value of features

    np.save(file, dataset_world.max)#max value of features
# Save Model properties to special numpy file with extension (.npy)
with open('/content/Optimal_thetas_of Neural_Network_with_Metadata_US.npy',
'wb') as file:

  np.save(file, model_US.no_of_layers) # save number of layers

  np.save(file, model_US.activation_fnt) # save activation function

  for layer in model_US.network:

    if layer != f'layer{model_US.no_of_layers}':

      np.save(file, model_US.network[layer][thetas]) #save thetas of each la
yer

#Append Data properties to the same numpy file
with open('/content/Optimal_thetas_of Neural_Network_with_Metadata_US.npy',
'ab') as file:

    np.save(file, dataset_US.remove_cols) #save column names to be removed f
rom the features

    np.save(file, dataset_US.scale) # Whether scaling is done on the dataset
 or not

    np.save(file, dataset_US.scale_type) #If scaling is done, which one is c
arried out

    np.save(file, dataset_US.min) #min value of features

    np.save(file, dataset_US.max)#max value of features

#Perform prediction on the test data
def prediction(model, data):

  #args:
  #     model: object of class Network
  #     data: (tuple) containing both X and Y as tuple e.g. (X, Y)


  #calculate output
  model.forward_propagate(data[0])
  predict= model.network[f'layer{model.no_of_layers}']['pred_outputs']

  #calculate cost
  cost=model.MSE(model.network[f'layer{model.no_of_layers}']['pred_outputs']
, data[1])
```

```
  #Scatter plot between predicted and actual values
  plt.plot(predict, 'ro', label='predicted')
  plt.plot(data[1],'bo', label='actual')
  plt.title("Predicted vs Actual")
  plt.legend()
  plt.axis()
  plt.show()

  return (predict, cost)
_, train_cost_world=prediction(model_world, train_data_world)
print(f'Cost on the train data is:{train_cost_world}')

_, valid_cost_world=prediction(model_world, valid_data_world)
print(f'Cost on the validation data is:{valid_cost_world}')

_, test_cost_world=prediction(model_world, test_data_world)
print(f'Cost on the test data is:{test_cost_world}')

_, train_cost_US=prediction(model_US, train_data_US)
print(f'Cost on the training data is:{train_cost_US}')

_, valid_cost_US=prediction(model_US, valid_data_US)
print(f'Cost on the validation data is:{valid_cost_US}')

_, test_cost_US=prediction(model_US, test_data_US)
print(f'Cost on the test data is:{test_cost_US}')
```

Optimal Thetas:

The thetas of the network learned for each dataset are given below:

For Worldwide:

```
The Optimal Thetas for the Worldwide Dataset are:
layer1
[[0.32799735 0.99706518 0.54608254 0.65058905 0.80004573 0.43057387]
 [1.08675656 0.82688933 0.71047115 0.37042483 1.02354338 1.01588517]
 [0.60150005 0.78264973 0.87106193 0.70470336 0.53804487 0.4314366 ]
 [0.38452969 0.21211353 0.7497386  0.62556321 0.1672801  0.31897397]
 [0.52676477 0.49461791 0.18140939 0.24892253 0.21903332 1.04608281]]
layer2
[[0.953138   1.30449229 0.51720485 1.09592136]
 [1.05251856 1.6956855  0.44791361 0.95842683]
 [1.6190002  1.60452356 0.85932185 1.41810701]
 [1.44264719 1.10686619 0.28703917 1.2957748 ]
 [1.31419272 1.13990613 0.85032351 0.51780276]
 [1.08878671 1.40601717 0.93227311 0.50232009]
 [0.79523408 1.49537856 0.96496559 0.98468761]]
layer3
[[16.76757668  9.9510781   7.65586708]
 [15.92172118 10.04788746  7.17046445]
 [14.45590585  8.86102893  7.00960315]
 [15.96577939  9.67604972  6.85118961]
 [16.22673396  9.43742737  7.01264265]]
layer4
[[364.04807708  58.14407978]
 [274.27599892  43.92948154]
 [279.8047426   44.78030354]
 [338.17679686  53.49477739]]
layer5
[[64143.21783645]
 [63374.77471187]
 [63690.14231861]]
```

For US:

```
The Optimal Thetas for the US Dataset are:
layer1
[[5.50672521 4.28421629 5.93981205]
 [3.16924959 1.96090024 3.00732055]
 [2.4529272  2.12704676 2.63856483]
 [4.28675542 2.97087904 3.67973578]
 [1.69908553 0.87693472 2.30869377]
 [1.26619252 0.80908262 0.94914656]]
layer2
[[  8.28773372   9.56282877   5.02304155  10.27854097 748.55611822]
 [  6.45344512   7.78956985   4.19539659   9.50550224 742.19398188]
 [  7.38341903   8.23927978   4.37249346   9.54825588 745.72330976]
 [  6.95085091   8.19680174   4.03400651   9.59489694 741.78653477]]
layer3
[[3864.55784746]
 [3826.74199005]
 [3818.50948644]
 [3835.92995024]
 [3831.08081103]
 [3832.92989375]]
```

Annex C: Prediction Code

```
import pandas as pd
import numpy as np
import sys

class Prediction:

  def __init__(self, dir_to_data, dir_to_thetas_metadata):

    #read  input data and remove the example (row) whose values are not prov
ided or contains NA.
    self.data=pd.read_csv(dir_to_data).dropna()

    #list for storing thetas of each layer
    self.thetas=[]

    #read meta data from the file in the same order as it was written to the
 file
    with open(dir_to_thetas_metadata, 'rb') as file:

      #read the total number of layers in the network
      self.no_of_layers = np.load(file)

      #read and use corresponding activation function
      activation_fnt = np.load(file, allow_pickle=True)
      if activation_fnt == 'Sigmoid':
        self.ActivationFnt = self.SigmoidFnt
      else:
        self.ActivationFnt = self.ReLU

      #read thetas
      for i in range(self.no_of_layers-1):
        self.thetas.append(np.load(file))

      #read column names which needs to be removed
      remove_cols = np.load(file)

      #remove columns not required in the features
      if len(remove_cols):
        self.data=self.data.drop(remove_cols, axis=1)

      #read data related to scaling
      self.scale=np.load(file)
      self.scale_type=np.load(file, allow_pickle=True)

      #read minimium and maximum values of the dataset used for creating mod
el
      self.min=np.load(file, allow_pickle=True)
      self.max=np.load(file, allow_pickle=True)

    #if number of columns of the input data (features) is not equal to the n
odes in the input layer
    if self.data.shape[1] != self.thetas[0].shape[0] -1:
      sys.exit(f'Features provided are either more or less than input nodes.
 Please check your input data file.')
```

```
  #Perform scaling of the features
  def scaleDataset(self):

    if self.scale_type: #between 0 and 1
      self.data = self.data.apply(lambda x: (x-self.min)/(self.max-self.min)
, axis=1)

    else: #between -1 and 1
      self.data = self.data.apply(lambda x: 2*((x-self.min)/(self.max-self.m
in))-1, axis=1)


  #Add bias to the activations as column vector
  def AddBias(self, X):
    return np.c_[ np.ones(X.shape[0]), X]

  #Apply linear function
  def LinearFnt(self, X, theta):
    return np.dot(X, theta)

  #Apply Sigmoid function
  def SigmoidFnt(self, Z):
    return 1/(1+(np.exp(-Z)))

  #Apply ReLU
  def ReLU(self, Z):
    return np.maximum(0, Z)

  #perform prediction on input data
  def prediction(self):

    #scale the dataset
    if self.scale:
      self.scaleDataset()

    self.data =self.data.to_numpy()

    #Forward Propagation
    for i in range(self.no_of_layers):

      if i == 0: #input layer
        activations=self.AddBias(self.data)

      elif i == self.no_of_layers -1: #output layer
        output=self.LinearFnt(activations, self.thetas[i-1]) #no activation
function

      else:
        activations =self.AddBias(self.ActivationFnt(self.LinearFnt(activati
ons, self.thetas[i-1])))

    print(f'The Predicted Values for the given inputs is: \n{output}')
    return output

#directories to the data
```

```
dir_to_data_world ='/content/World_Data_Template_For_Prediction_Code.csv'
dir_to_thetas_metadata_world='/content/Optimal_thetas_of Neural_Network_with
_Metadata_Worldwide.npy'

pred_world=Prediction(dir_to_data_world, dir_to_thetas_metadata_world)
_=pred_world.prediction()

#directories to the data
dir_to_data_US ='/content/US_Data_Template_For_Prediction_Code.csv'
dir_to_thetas_metadata_US='/content/Optimal_thetas_of Neural_Network_with_Me
tadata_US.npy'

pred_US=Prediction(dir_to_data_US, dir_to_thetas_metadata_US)
_=pred_US.prediction()
```