Deep-Learning based model on Travel-Time Estimation

REPORT

Abstract:

Smart cities can effectively improve the quality of urban life. But, with the rise in population, there is an increased demand for vehicles for transportations. This becomes an obvious reason for traffic jams, congestions etc. Intelligent Transportation System (ITS) is an important part of smart cities. The accurate and real-time prediction of traffic flow plays an important role in ITSs. Therefore, we come forth and present a model that helps us in traffic prediction and gives close to accurate travel time estimation from one location(node) to another. We aim to provide an in-depth comprehension on traffic using deep learning-based tools, thus providing proper solution to the in-hand problem, using the advance technologies to extract the real time data and adopting it to our advantage.

Introduction:

Travel Time Estimation (TTE) module can allow us to perceive the urban traffic situation in advance, and dispatch or control vehicles in real-time based on the feedback information to prevent large-scale congestion.

Traditionally, some classic **time series modelling** models and some data-driven **statistical learning models** were used in this field, especially some ensemble regression tree models.

Although these traditional methods are simple, intuitive and computationally light, they cannot integrate more complex spatial and temporal dynamics, which limits the improvement of accuracy. In recent years, deep learning models have become the fruitful approach for spatial–temporal learning and inference. Although some deep learning models have achieved some breakthroughs, there are still some limitations to be improved. Most of previous deep learning models in TTE are traditional like-

- **Convolutional Neural Networks (CNNs)** and their variants are adopted to extract grid spatial features while
- **Recurrent Neural Networks (RNNs)** and their variants are introduced into capturing temporal dynamics.

We combine these two different scale models for spatial–temporal representation learning, which can be successfully extended to Travel Time Estimation (TTE). However, such methods can have certain drawbacks.

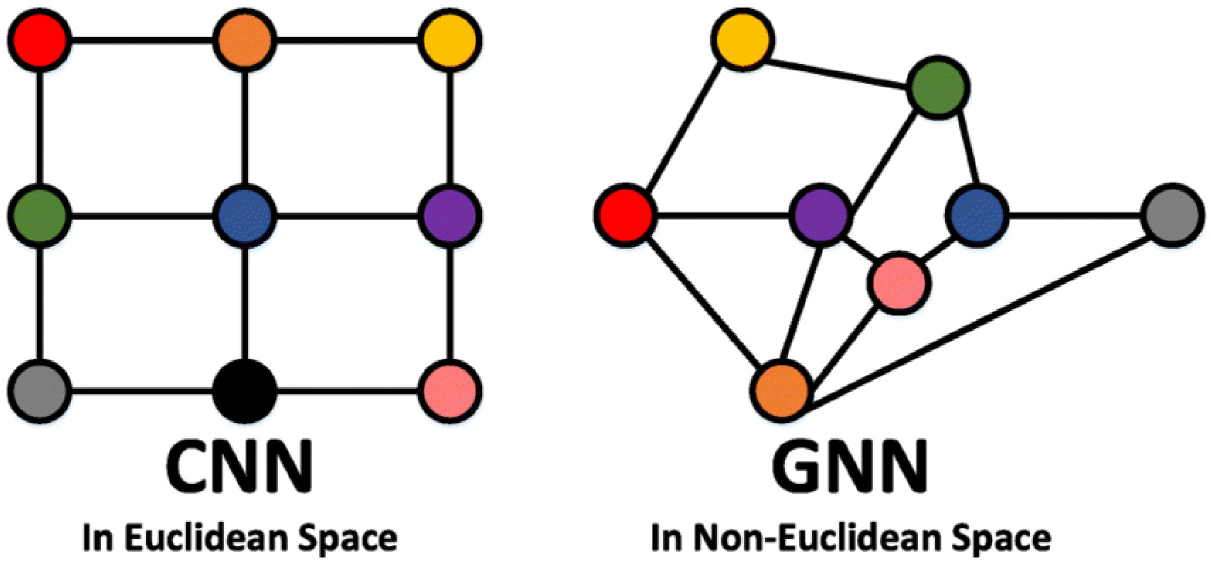
Disadvantages of single deep models or hybrid CNN-RNN based models

- These models ignore the topology structure of traffic network, which leads to the inability to learn fine-grained spatial–temporal representation.
- Second, most of previous works are difficult to capture multi-scale spatial–temporal dynamics in traffic networks. Many deep learning frameworks capture the high-level spatial– temporal dynamics by stacking multiple layers, but ignore the impact of shallow-level information at different scales on high-level information.

Advantages of Graph Convolutional Network (GCN)

In the beginning, most of the previous works related to interconnected topological networks were based on the **grid maps** for exploration. The limitation of grid map modelling method can be foreseen when dealing with **non-Euclidean spaces**. Graph Convolutional Networks (GCN), helps us resolve this issue. Graph convolution network **(GCN)** is the promotion of normal convolution network, which can handle the spatial representation learning in non-Euclidean structure.

With the development of **graph deep learning**, Graph Neural Networks (**GNNs**) or Graph Convolutional Networks (**GCNs**) have become the most effective methods for representation learning in spatial topology structure.



Although our entire model revolves around graph deep learning, we have incorporated other methods like **Long Short-Term Memory (LSTM)** and **Autoencoder** for feature vector generation along with **hierarchical clustering** to group and process the data accordingly, the working of which is explained further.

Working:

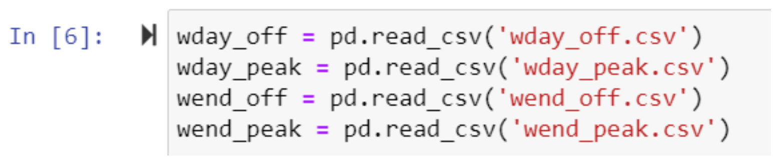
Real-World Data collected and filtered from the taxi trajectories of over 14 thousand taxis in Chengdu (city in China) is used to train and test our proposed model. Statistics and graph attributes of our dataset are as follows-

Latitude range	[30.66, 30.68]
Longitude range	[104.06, 104.08]
Number of trajectories	13442
Number of links	532
Average travel time (s)	246.54
Average moving distance (m)	1417.23
Average Degree	2.5582
Density	0.0048
Clustering coefficient	0.0125

Extracting Temporal Data:-

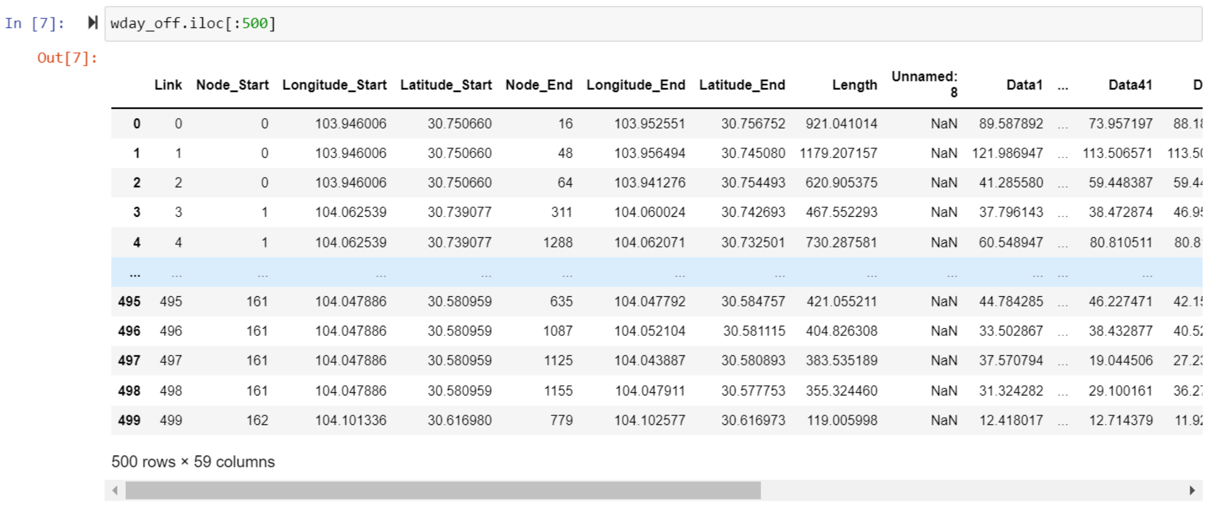
To use the autoencoder, we need to first extract the temporal features from the given dataset. Here, our temporal features are the time taken by 50 individual cabs from one node to another.

Importing Data to Jupyter notebook



```
In [6]:  wday_off = pd.read_csv('wday_off.csv')
         wday_peak = pd.read_csv('wday_peak.csv')
         wend_off = pd.read_csv('wend_off.csv')
         wend_peak = pd.read_csv('wend_peak.csv')
```

Initial Dataset



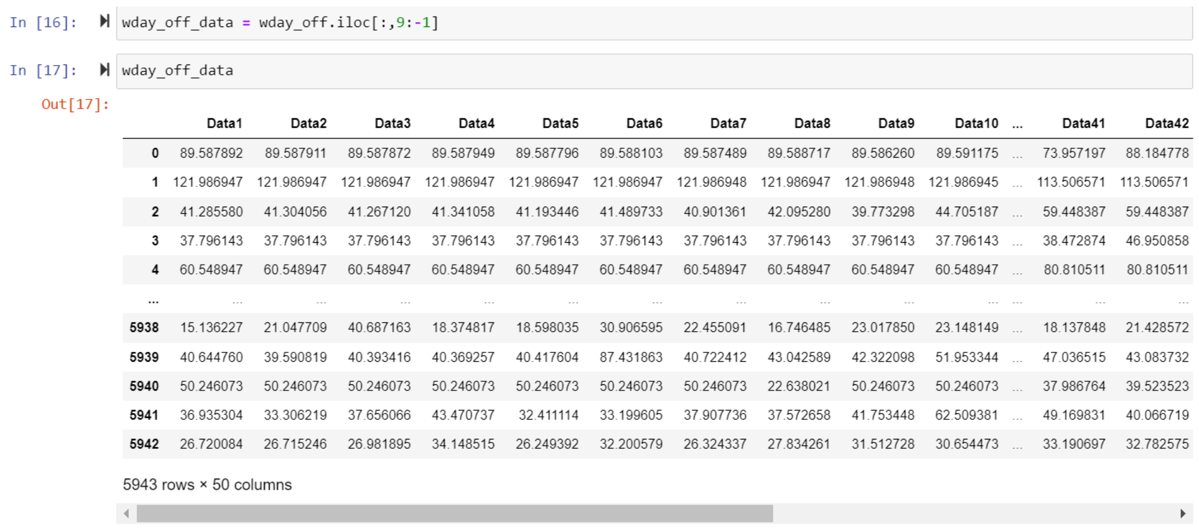
```
In [7]:  wday_off.iloc[:500]
```

Out[7]:

	Link	Node_Start	Longitude_Start	Latitude_Start	Node_End	Longitude_End	Latitude_End	Length	Unnamed: 8	Data1	...	Data41	D
0	0	0	103.946006	30.750660	16	103.952551	30.756752	921.041014	NaN	89.587892	...	73.957197	88.1
1	1	0	103.946006	30.750660	48	103.956494	30.745080	1179.207157	NaN	121.986947	...	113.506571	113.5
2	2	0	103.946006	30.750660	64	103.941276	30.754493	620.905375	NaN	41.285580	...	59.448387	59.4
3	3	1	104.062539	30.739077	311	104.060024	30.742693	467.552293	NaN	37.796143	...	38.472874	46.9
4	4	1	104.062539	30.739077	1288	104.062071	30.732501	730.287581	NaN	60.548947	...	80.810511	80.8
...	...	...	...	...	...	...	...	...	...	...	...	...	
495	495	161	104.047886	30.580959	635	104.047792	30.584757	421.055211	NaN	44.784285	...	46.227471	42.1
496	496	161	104.047886	30.580959	1087	104.052104	30.581115	404.826308	NaN	33.502867	...	38.432877	40.5
497	497	161	104.047886	30.580959	1125	104.043887	30.580893	383.535189	NaN	37.570794	...	19.044506	27.2
498	498	161	104.047886	30.580959	1155	104.047911	30.577753	355.324460	NaN	31.324282	...	29.100161	36.2
499	499	162	104.101336	30.616980	779	104.102577	30.616973	119.005998	NaN	12.418017	...	12.714379	11.9

500 rows × 59 columns

Temporal Data



```
In [16]:  wday_off_data = wday_off.iloc[:,9:-1]
```

```
In [17]:  wday_off_data
```

Out[17]:

	Data1	Data2	Data3	Data4	Data5	Data6	Data7	Data8	Data9	Data10	...	Data41	Data42
0	89.587892	89.587911	89.587872	89.587949	89.587796	89.588103	89.587489	89.588717	89.586260	89.591175	...	73.957197	88.184778
1	121.986947	121.986947	121.986947	121.986947	121.986947	121.986947	121.986948	121.986947	121.986948	121.986945	...	113.506571	113.506571
2	41.285580	41.304056	41.267120	41.341058	41.193446	41.489733	40.901361	42.095280	39.773298	44.705187	...	59.448387	59.448387
3	37.796143	37.796143	37.796143	37.796143	37.796143	37.796143	37.796143	37.796143	37.796143	37.796143	...	38.472874	46.950858
4	60.548947	60.548947	60.548947	60.548947	60.548947	60.548947	60.548947	60.548947	60.548947	60.548947	...	80.810511	80.810511
...	...	...	...	...	...	...	...	...	...	...	...	...	...
5938	15.136227	21.047709	40.687163	18.374817	18.598035	30.906595	22.455091	16.746485	23.017850	23.148149	...	18.137848	21.428572
5939	40.644760	39.590819	40.393416	40.369257	40.417604	87.431863	40.722412	43.042589	42.322098	51.953344	...	47.036515	43.083732
5940	50.246073	50.246073	50.246073	50.246073	50.246073	50.246073	50.246073	22.638021	50.246073	50.246073	...	37.986764	39.523523
5941	36.935304	33.306219	37.656066	43.470737	32.411114	33.199605	37.907736	37.572658	41.753448	62.509381	...	49.169831	40.066719
5942	26.720084	26.715246	26.981895	34.148515	26.249392	32.200579	26.324337	27.834261	31.512728	30.654473	...	33.190697	32.782575

5943 rows × 50 columns

AutoEncoder and Long Short-Term Memory (LSTM)

An **Autoencoder** is a neural network which is an unsupervised learning algorithm which uses back propagation to generate output value which is almost close to the input value. It takes input such as image or vector anything with a very high dimensionality and run through the neural network and tries to compress the data into a smaller representation with two principal components. The first one is the **encoder** which is simply a bunch of layers that are full connected layers or convolutional layers which are going to take the input and compress it to a smaller representation which has less dimensions than the input which is known as **bottleneck**. Now from this bottleneck it tries to reconstruct the input using full connected layers or convolutional layers.

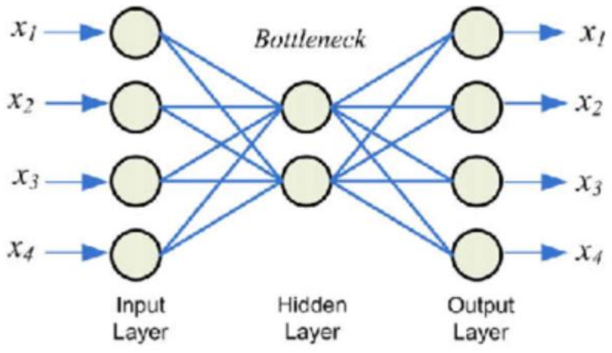


Figure 1: Layers in an autoencoder [1]

Autoencoders are a type of feed-forward network that may be trained using the same procedures as feed-forward networks. The output of the Autoencoder is the same as the input with some loss. Thus, autoencoders are also called **lossy compression technique**. Data denoising and dimensionality reduction for data visualization are the two major applications of autoencoders.

Some datasets have a complex relationship within the features. Thus, using only one Autoencoder is not sufficient. A single Autoencoder might be unable to reduce the dimensionality of the input features. Therefore, for such use cases, we use **stacked**

autoencoders. The stacked autoencoders are, as the name suggests, multiple encoders stacked on top of one another.

Before discussing about stacked autoencoder in detail, we need to have a brief overview of sparse autoencoder.

Sparse Autoencoder: An autoencoder takes the input image or vector and learns code dictionary that changes the raw input from one representation to another. Where in sparse autoencoders with a sparsity enforcer that directs a single layer network to learn code dictionary which in turn minimizes the error in reproducing the input while restricting number of code words for reconstruction.

The sparse autoencoder consists a single hidden layer, which is connected to the input vector by a weight matrix forming the encoding step. The hidden layer then outputs to a reconstruction vector, using a tied weight matrix to form the decoder.

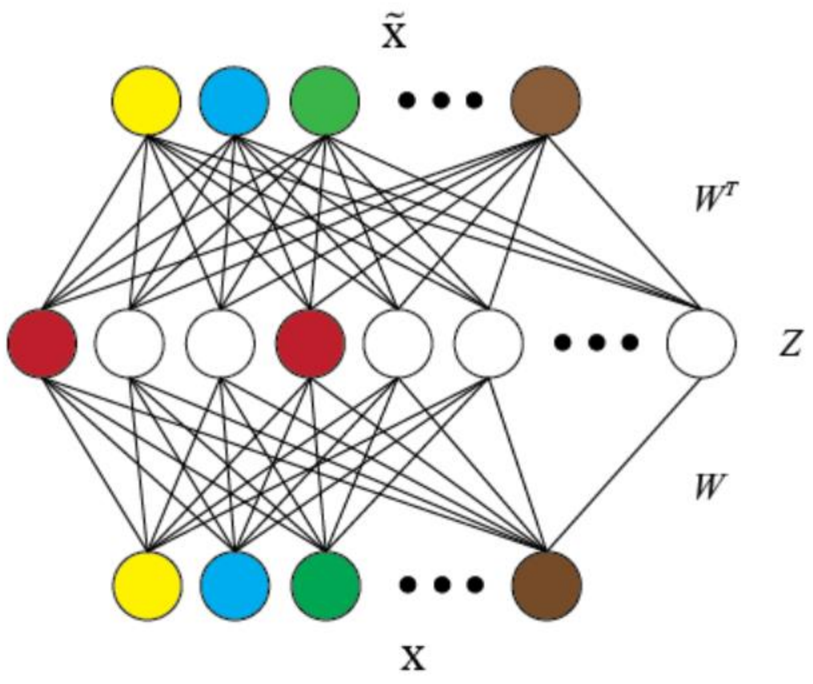


Figure 2: Sparse autoencoder[8]

Stacked Autoencoder:

A stacked autoencoder is a neural network consist several layers of sparse autoencoders where output of each hidden layer is connected to the input of the successive hidden layer.

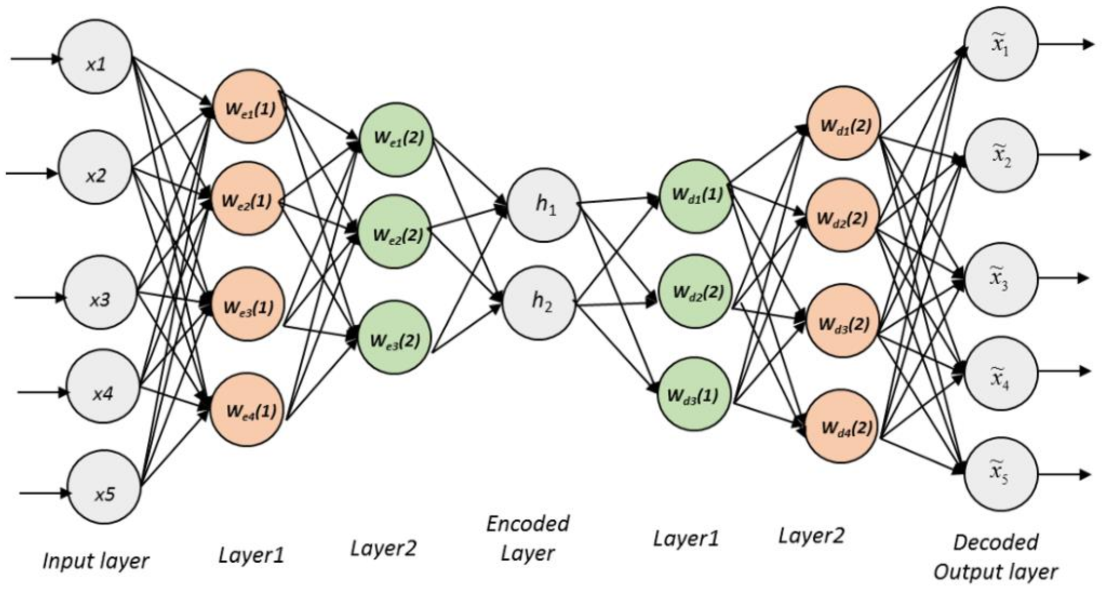


Figure 3: Stacked Autoencoder[3]

As shown in Figure above the hidden layers are trained by an unsupervised algorithm and then fine-tuned by a supervised method. Stacked autoencoder mainly consists of three steps:

1. Train autoencoder using input data and acquire the learned data.

2. The learned data from the previous layer is used as an input for the next layer and this continues until the training is completed.

3. Once all the hidden layers are trained use the backpropagation algorithm to minimize the cost function and weights are updated with the training set to achieve fine tuning.

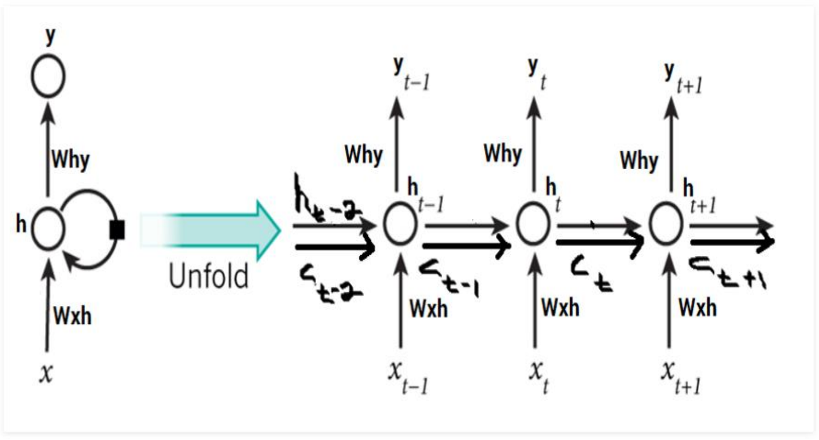
In the architecture of the stacked autoencoder, the layers are typically symmetrical with regards to the central hidden layer.

The recent advancements in Stacked Autoencoder is it provides a version of raw data with much detailed and promising feature information, which is used to train a classier with a specific context and find better accuracy than training with raw data.

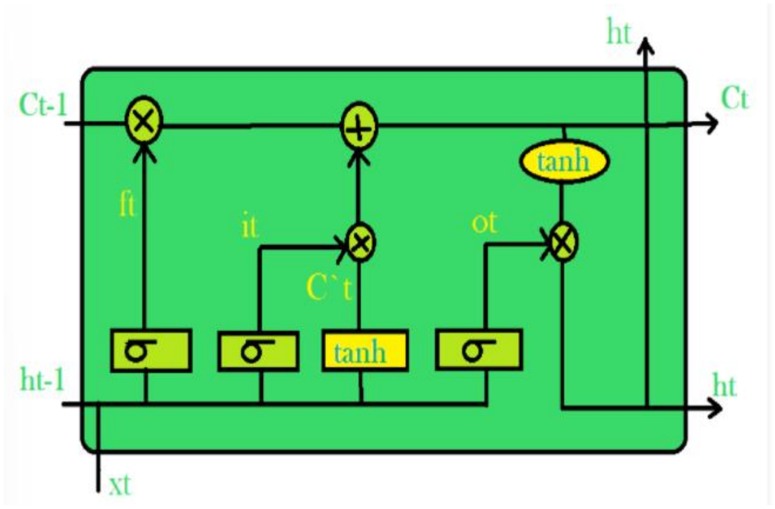
Long Short-Term Memory is an advanced version of recurrent neural network (RNN) architecture that was designed to model chronological sequences and their long-range dependencies more precisely than conventional RNNs.

The hidden layer of LSTM is a gated unit or gated cell. It consists of four layers that interact with one another in a way to produce the output of that cell along with the cell state. These two things are then passed onto the next hidden layer. Unlike RNNs which have got the only single neural net layer of tanh, LSTMs comprises of three logistic sigmoid gates and one tanh layer. Gates have been introduced in order to limit the information that is passed through the cell. They determine which part of the information will be needed by the next cell and which part is to be discarded. The output is usually in the range of 0-1 where ‘0’ means ‘reject all’ and ‘1’ means ‘include all’.

Hidden layers of LSTM :



The second sigmoid layer is the input gate that decides what new information is to be added to the cell. It takes two inputs h_{t-1} and x_t. The tanh layer creates a vector C_t of the new candidate values. Together, these two layers determine the information to be stored in the cell state. Their point-wise multiplication tells us the amount of information to be added to the cell state. The result is then added with the result of the forget gate multiplied with previous cell state f_t*C_{t-1} to produce the current cell state C_t. Next, the output of the cell is calculated using a sigmoid and a tanh layer. The sigmoid layer decides which part of the cell state will be present in the output whereas tanh layer shifts the output in the range of [-1,1]. The results of the two layers undergo point-wise multiplication to produce the output h_t of the cell.



Benefits of using LSTM

During the training process of a network, the main goal is to minimize loss (in terms of error) observed in the output when training data is sent through it. We calculate the **gradient**, that is, loss with respect to a particular set of weights, adjust the weights accordingly and repeat this process until we get an optimal set of weights for which loss is minimum. This is the concept of backtracking. Sometimes, it so happens that the gradient is almost negligible. It must be noted that the gradient of a layer depends on certain components in the successive layers. If some of these components are small (less than 1), the result obtained, which is the gradient, will be even smaller. This is known as the **scaling effect**. When this gradient is multiplied with the learning rate which is in itself a small value ranging between 0.1-0.001, it results in a smaller value. As a consequence, the alteration in weights is quite small, producing almost the same output as before. Similarly, if the gradients are quite large in value due to the large values of components, the weights get updated to a value beyond the optimal value. This is known as the problem of **exploding gradients**. To avoid this scaling effect, the neural network unit was re-built in such a way that the scaling factor was fixed to one. The cell was then enriched by several gating units and was called **LSTM**.

Use of AutoEncoder and LSTM in Travel Time Estimation

In our model, AutoEncoder is used for feature extraction and LSTM model can be used for data prediction.

The AutoEncoder is a kind of unsupervised learning, which can be used as **feature extractor** of our **temporal data**. We determine the initial value of the weight matrix before the training. The weight matrix plays a very important role in the network. we hope to retain the characteristics of the original data after training the weight matrix. If the extracted feature can reconstruct the original data well, it indicates that the features of the original data can be effectively retained through the weight matrix.

Given an original input sequence data $X = \{x_1, x_2, . . ., x_k\}$, where $xi \in R^d$. The characteristic sequence of the original data is obtained through the formula f. T represents the characteristic sequence of the original sequence X, which is defined as $T = \{t_1, t_2, . . . , t_k\}$, where $t_i \in R^l$. The output of the encoder is used as the input of the decoder. The decoder reconstructs the original data according to the characteristic sequence T. The reconstructed data $Y = \{y_1, y_2, . . . , y_k\}$, where $y_i \in R^d$. The purpose of decoding is to verify whether the extracted features are valid. After the training of the AutoEncoder is completed, we only use the encoder to extract the characteristics of the original data to obtain more internal structure of the data.

The encoding and decoding process follows the equations:

$$t_i = f(w_t \cdot x_i + b_t),$$

$$y_i = g(w_y \cdot t_i + b_y)$$

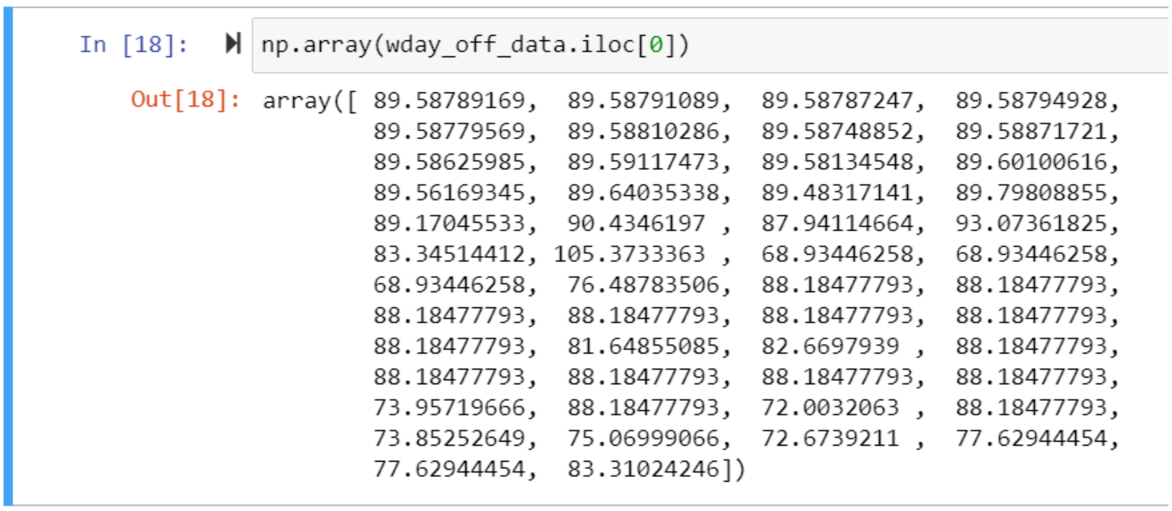
where f (·) and g (·) are the sigmoid functions, and wt , wy and bt , by are weights and biases, respectively. We train the AutoEncoder by minimizing reconstruction error

$$L(X,Y) = \frac{1}{2}\sum_{i=1}^{n} |x_i - y_i|^2$$

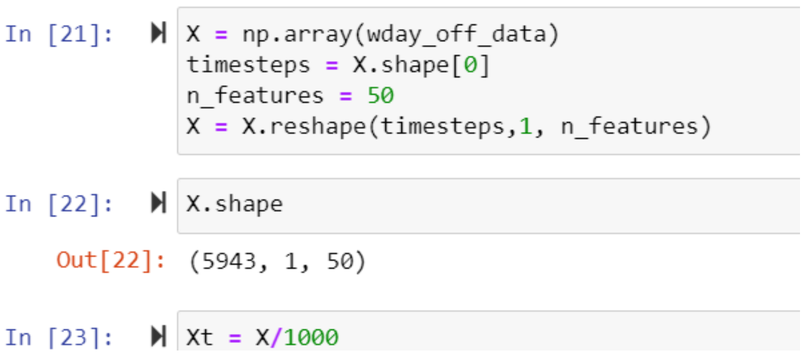
When the difference between the reconstructed data Y and the original data X is small enough, in other words, the output T of the coding process is valid, T is seen as the characteristics extracted from the original data.

Implementation

Sequencing of the temporal data-

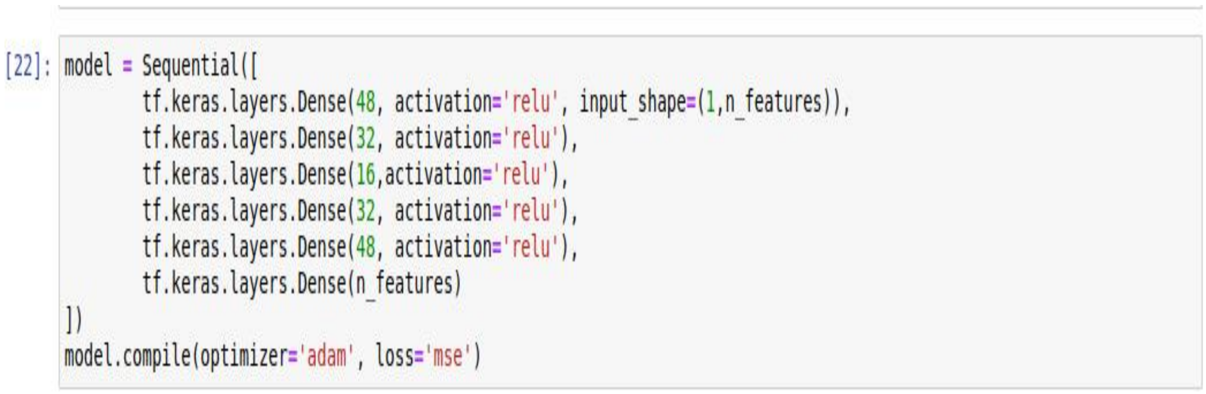


```
In [18]:   np.array(wday_off_data.iloc[0])

Out[18]: array([ 89.58789169,  89.58791089,  89.58787247,  89.58794928,
                 89.58779569,  89.58810286,  89.58748852,  89.58871721,
                 89.58625985,  89.59117473,  89.58134548,  89.60100616,
                 89.56169345,  89.64035338,  89.48317141,  89.79808855,
                 89.17045533,  90.4346197 ,  87.94114664,  93.07361825,
                 83.34514412, 105.3733363 ,  68.93446258,  68.93446258,
                 68.93446258,  76.48783506,  88.18477793,  88.18477793,
                 88.18477793,  88.18477793,  88.18477793,  88.18477793,
                 88.18477793,  81.64855085,  82.6697939 ,  88.18477793,
                 88.18477793,  88.18477793,  88.18477793,  88.18477793,
                 73.95719666,  88.18477793,  72.0032063 ,  88.18477793,
                 73.85252649,  75.06999066,  72.6739211 ,  77.62944454,
                 77.62944454,  83.31024246])
```

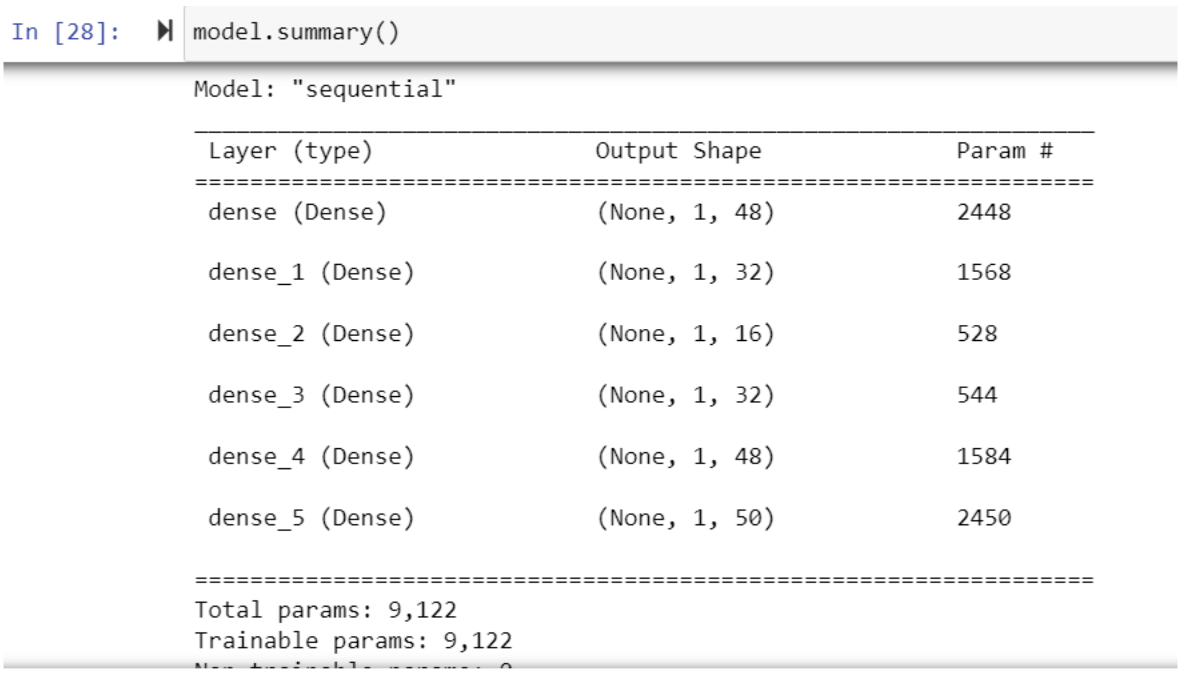


```
In [21]:   X = np.array(wday_off_data)
           timesteps = X.shape[0]
           n_features = 50
           X = X.reshape(timesteps,1, n_features)

In [22]:   X.shape

Out[22]: (5943, 1, 50)

In [23]:   Xt = X/1000
```

Autoencoder model to extract most important features from temporal data. Data reduced from 50 features to 16

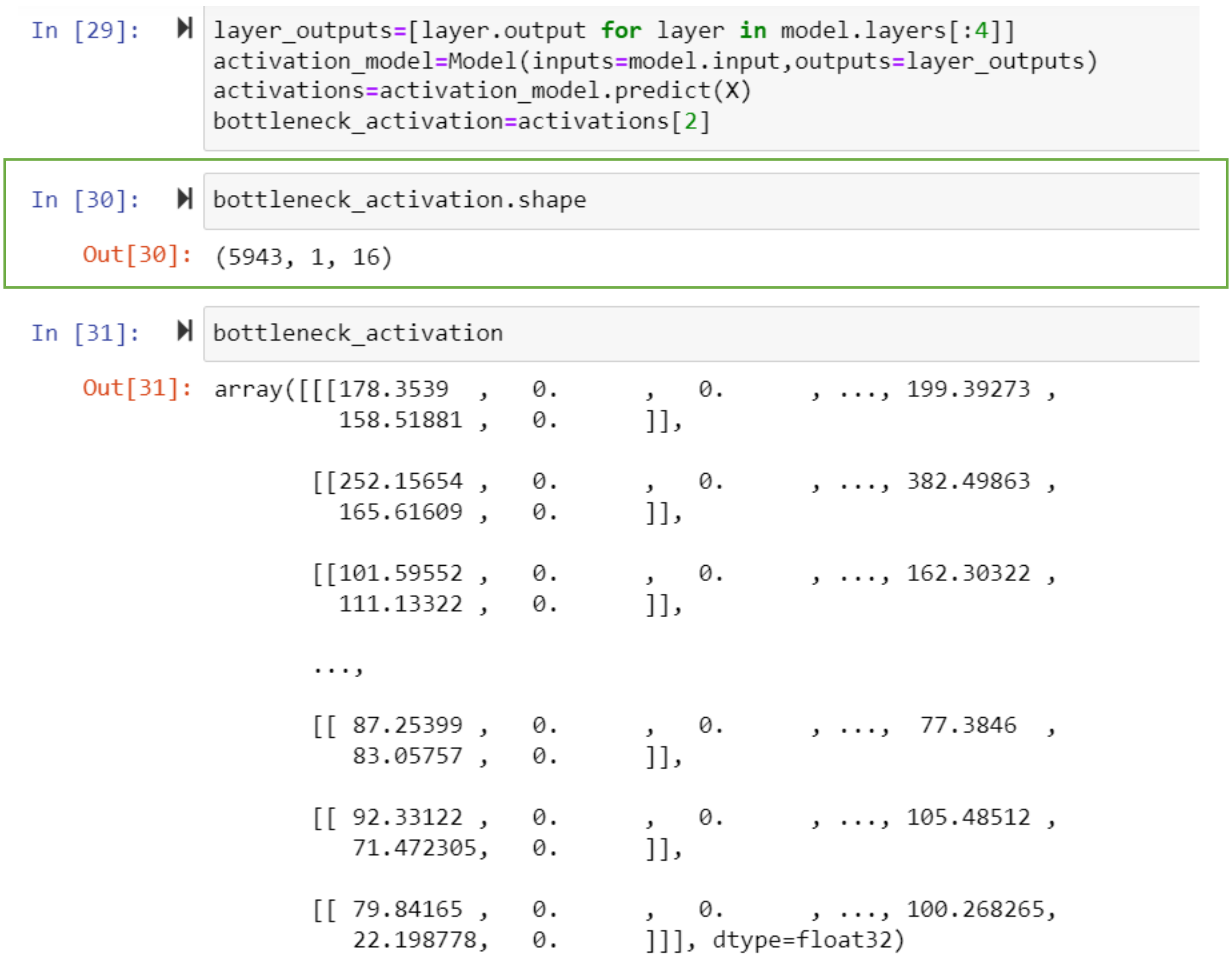


```
[22]: model = Sequential([
              tf.keras.layers.Dense(48, activation='relu', input_shape=(1,n_features)),
              tf.keras.layers.Dense(32, activation='relu'),
              tf.keras.layers.Dense(16,activation='relu'),
              tf.keras.layers.Dense(32, activation='relu'),
              tf.keras.layers.Dense(48, activation='relu'),
              tf.keras.layers.Dense(n_features)
      ])
      model.compile(optimizer='adam', loss='mse')
```



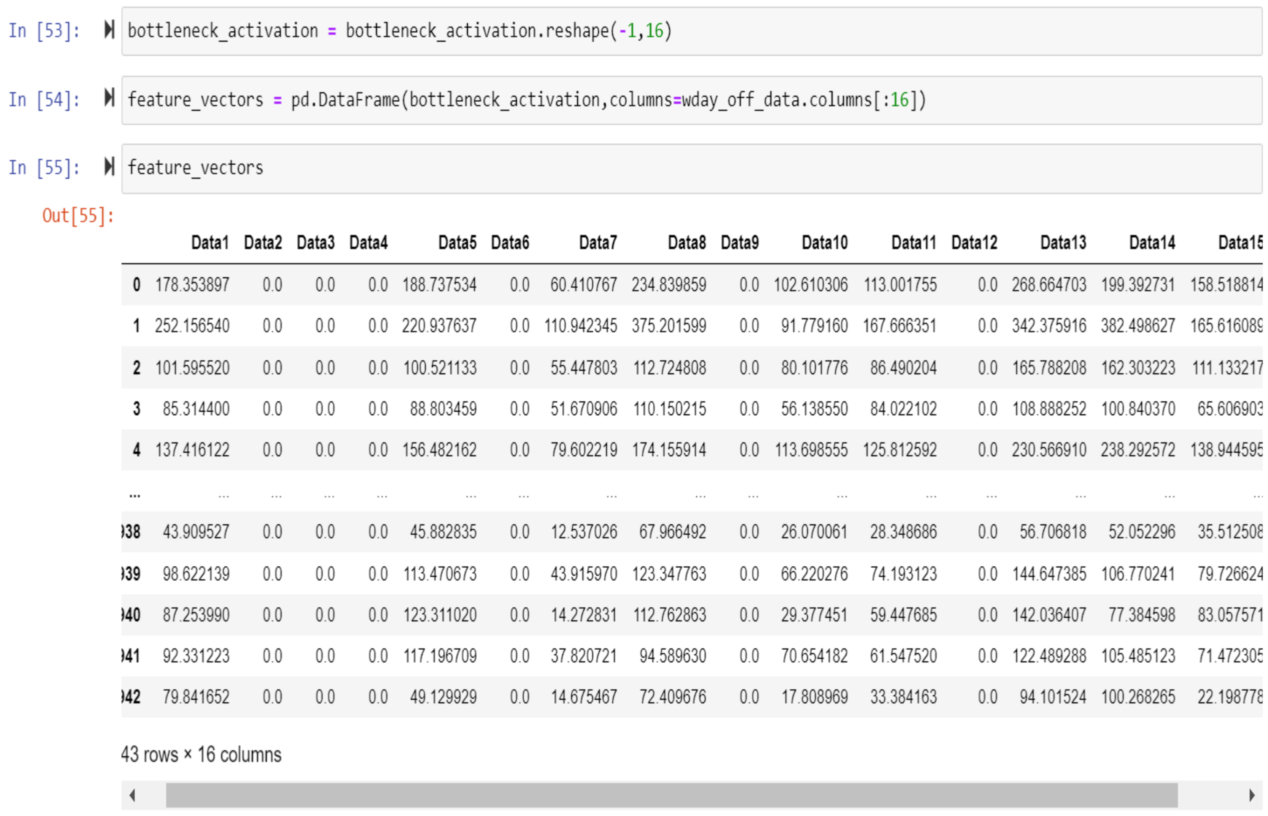
```
In [28]:  model.summary()

          Model: "sequential"
          _________________________________________________________________
           Layer (type)                Output Shape              Param #
          =================================================================
           dense (Dense)               (None, 1, 48)             2448

           dense_1 (Dense)             (None, 1, 32)             1568

           dense_2 (Dense)             (None, 1, 16)             528

           dense_3 (Dense)             (None, 1, 32)             544

           dense_4 (Dense)             (None, 1, 48)             1584

           dense_5 (Dense)             (None, 1, 50)             2450

          =================================================================
          Total params: 9,122
          Trainable params: 9,122
          Non-trainable params: 0
```

Bottleneck activation-



```
In [29]:  layer_outputs=[layer.output for layer in model.layers[:4]]
          activation_model=Model(inputs=model.input,outputs=layer_outputs)
          activations=activation_model.predict(X)
          bottleneck_activation=activations[2]

In [30]:  bottleneck_activation.shape

Out[30]:  (5943, 1, 16)

In [31]:  bottleneck_activation

Out[31]:  array([[[178.3539  ,   0.      ,   0.      , ..., 199.39273 ,
            158.51881 ,   0.      ]],

          [[252.15654 ,   0.      ,   0.      , ..., 382.49863 ,
            165.61609 ,   0.      ]],

          [[101.59552 ,   0.      ,   0.      , ..., 162.30322 ,
            111.13322 ,   0.      ]],

          ...,

          [[ 87.25399 ,   0.      ,   0.      , ...,  77.3846  ,
             83.05757 ,   0.      ]],

          [[ 92.33122 ,   0.      ,   0.      , ..., 105.48512 ,
             71.472305,   0.      ]],

          [[ 79.84165 ,   0.      ,   0.      , ..., 100.268265,
             22.198778,   0.      ]]], dtype=float32)
```

Generated feature vectors



```
In [53]:  bottleneck_activation = bottleneck_activation.reshape(-1,16)
```

```
In [54]:  feature_vectors = pd.DataFrame(bottleneck_activation,columns=wday_off_data.columns[:16])
```

```
In [55]:  feature_vectors
```

Out[55]:

	Data1	Data2	Data3	Data4	Data5	Data6	Data7	Data8	Data9	Data10	Data11	Data12	Data13	Data14	Data15
0	178.353897	0.0	0.0	0.0	188.737534	0.0	60.410767	234.839859	0.0	102.610306	113.001755	0.0	268.664703	199.392731	158.518814
1	252.156540	0.0	0.0	0.0	220.937637	0.0	110.942345	375.201599	0.0	91.779160	167.666351	0.0	342.375916	382.498627	165.616089
2	101.595520	0.0	0.0	0.0	100.521133	0.0	55.447803	112.724808	0.0	80.101776	86.490204	0.0	165.788208	162.303223	111.133217
3	85.314400	0.0	0.0	0.0	88.803459	0.0	51.670906	110.150215	0.0	56.138550	84.022102	0.0	108.888252	100.840370	65.606903
4	137.416122	0.0	0.0	0.0	156.482162	0.0	79.602219	174.155914	0.0	113.698555	125.812592	0.0	230.566910	238.292572	138.944595
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
38	43.909527	0.0	0.0	0.0	45.882835	0.0	12.537026	67.966492	0.0	26.070061	28.348686	0.0	56.706818	52.052296	35.512508
39	98.622139	0.0	0.0	0.0	113.470673	0.0	43.915970	123.347763	0.0	66.220276	74.193123	0.0	144.647385	106.770241	79.726624
40	87.253990	0.0	0.0	0.0	123.311020	0.0	14.272831	112.762863	0.0	29.377451	59.447685	0.0	142.036407	77.384598	83.057571
41	92.331223	0.0	0.0	0.0	117.196709	0.0	37.820721	94.589630	0.0	70.654182	61.547520	0.0	122.489288	105.485123	71.472305
42	79.841652	0.0	0.0	0.0	49.129929	0.0	14.675467	72.409676	0.0	17.808969	33.384163	0.0	94.101524	100.268265	22.198778

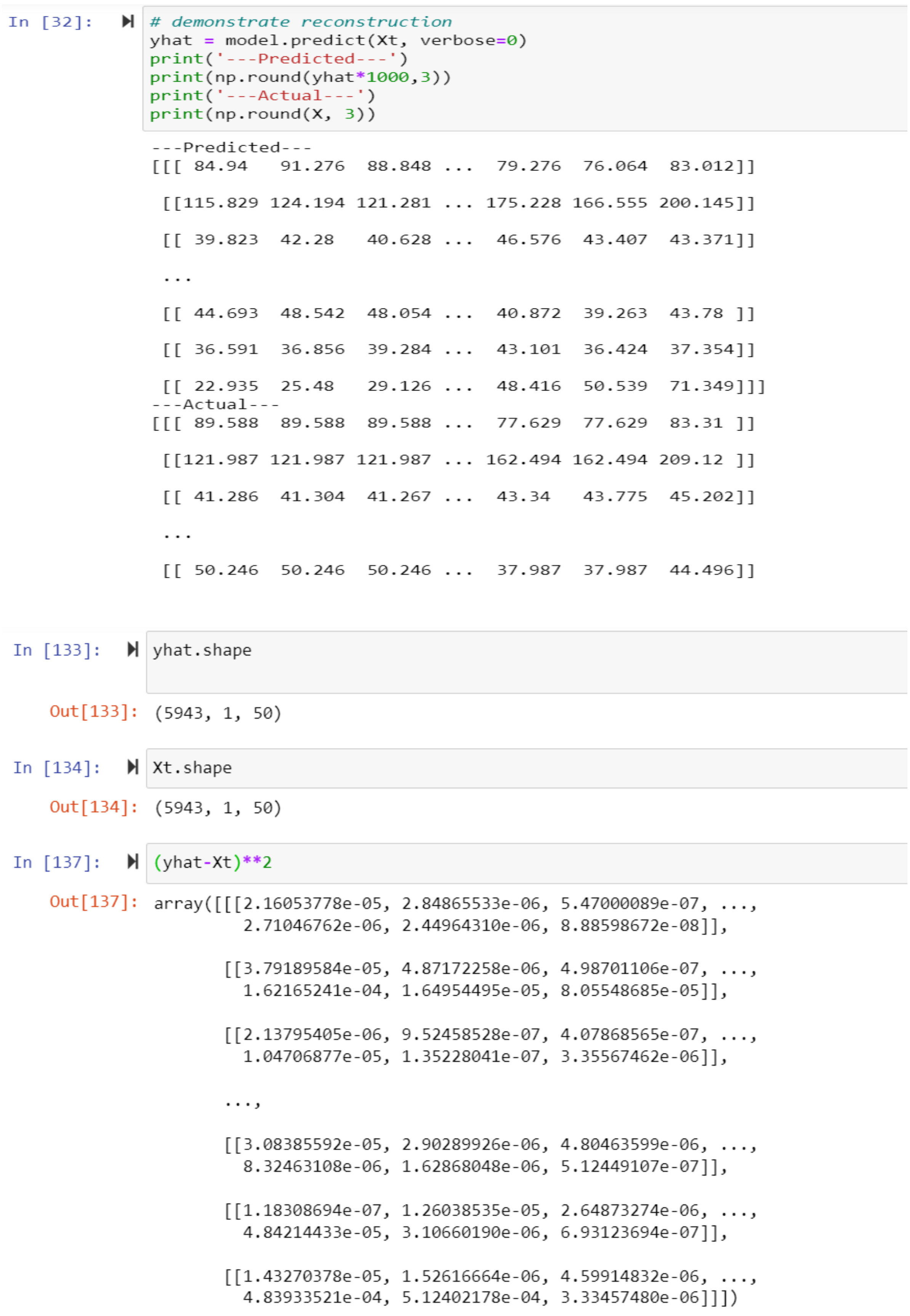
43 rows × 16 columns

Justification for the reduction in features-

Dimensionality is the number of input variables or features for a dataset. Dimensionality Reduction is the process of reducing the number of dimensions in the data either by excluding less useful features (Feature Selection) or transform the data into lower dimensions (Feature Extraction). Dimensionality reduction prevents overfitting. Overfitting is a phenomenon in which the model learns too well from the training dataset and fails to generalize well for unseen real-world data. Also, a lot of input features makes predictive modelling a more challenging task.

The most important thing to note here is that during the entire process, there should not be any information loss. The Autoencoder model has a tendency to generate the output similar to the input. We get the encoder layer and use the method predict to reduce dimensions in data. Since we have sixteen hidden units in the bottleneck the data is reduced to sixteen features. The bottleneck layer encodes into the simplest form, with only the important features being encoded, and thus there is no information loss.

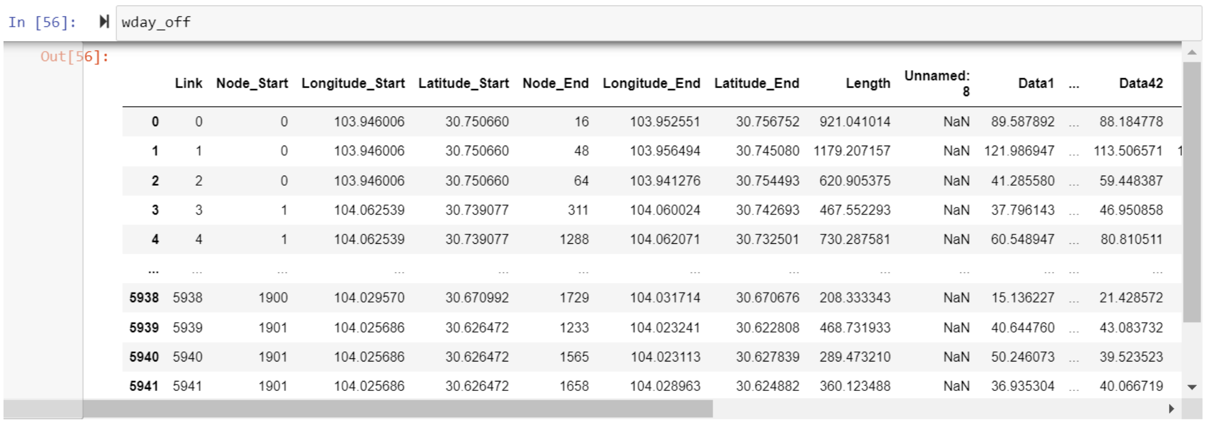
We can prove the same by taking the square of the difference between the features before and after feature extraction using autoencoder, which is infinitesimally small (of 10^{-5} order).



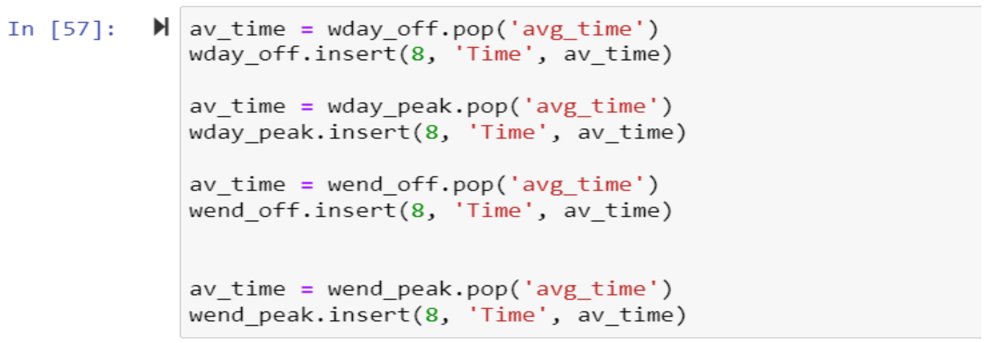
```
In [32]:  # demonstrate reconstruction
          yhat = model.predict(Xt, verbose=0)
          print('---Predicted---')
          print(np.round(yhat*1000,3))
          print('---Actual---')
          print(np.round(X, 3))

          ---Predicted---
          [[[ 84.94    91.276   88.848  ...   79.276   76.064   83.012]]

           [[115.829  124.194  121.281  ...  175.228  166.555  200.145]]

           [[ 39.823   42.28    40.628  ...   46.576   43.407   43.371]]

           ...

           [[ 44.693   48.542   48.054  ...   40.872   39.263   43.78 ]]

           [[ 36.591   36.856   39.284  ...   43.101   36.424   37.354]]

           [[ 22.935   25.48    29.126  ...   48.416   50.539   71.349]]]
          ---Actual---
          [[[ 89.588   89.588   89.588  ...   77.629   77.629   83.31 ]]

           [[121.987  121.987  121.987  ...  162.494  162.494  209.12 ]]

           [[ 41.286   41.304   41.267  ...   43.34    43.775   45.202]]

           ...

           [[ 50.246   50.246   50.246  ...   37.987   37.987   44.496]]


In [133]:  yhat.shape

Out[133]:  (5943, 1, 50)

In [134]:  Xt.shape

Out[134]:  (5943, 1, 50)

In [137]:  (yhat-Xt)**2

Out[137]:  array([[[2.16053778e-05, 2.84865533e-06, 5.47000089e-07, ...,
                    2.71046762e-06, 2.44964310e-06, 8.88598672e-08]],

                  [[3.79189584e-05, 4.87172258e-06, 4.98701106e-07, ...,
                    1.62165241e-04, 1.64954495e-05, 8.05548685e-05]],

                  [[2.13795405e-06, 9.52458528e-07, 4.07868565e-07, ...,
                    1.04706877e-05, 1.35228041e-07, 3.35567462e-06]],

                  ...,

                  [[3.08385592e-05, 2.90289926e-06, 4.80463599e-06, ...,
                    8.32463108e-06, 1.62868048e-06, 5.12449107e-07]],

                  [[1.18308694e-07, 1.26038535e-05, 2.64873274e-06, ...,
                    4.84214433e-05, 3.10660190e-06, 6.93123694e-07]],

                  [[1.43270378e-05, 1.52616664e-06, 4.59914832e-06, ...,
                    4.83933521e-04, 5.12402178e-04, 3.33457480e-06]]])
```

Now, we move on to the spatial features-

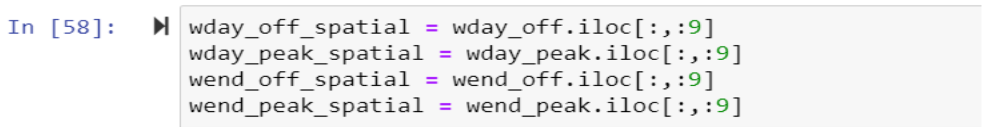
Extraction of spatial data from the original dataset



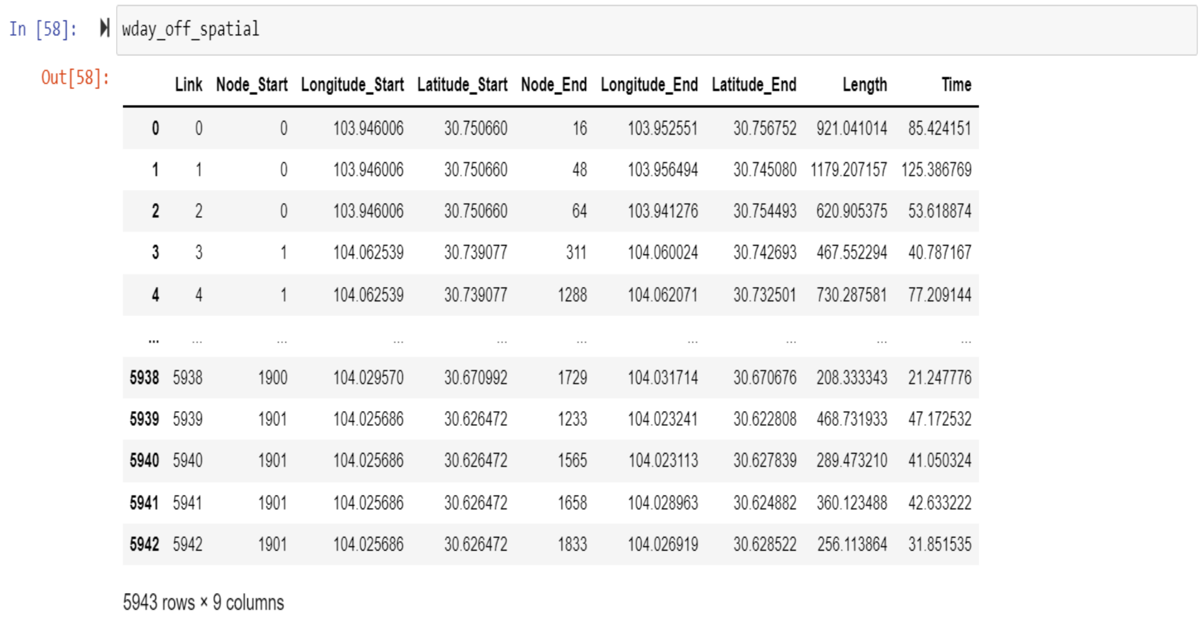
```
In [56]:  wday_off

Out[56]:
      Link  Node_Start  Longitude_Start  Latitude_Start  Node_End  Longitude_End  Latitude_End       Length  Unnamed:        Data1  ...      Data42
                                                                                                                  8
   0     0           0       103.946006       30.750660        16     103.952551     30.756752   921.041014       NaN    89.587892  ...   88.184778
   1     1           0       103.946006       30.750660        48     103.956494     30.745080  1179.207157       NaN   121.986947  ...  113.506571  1
   2     2           0       103.946006       30.750660        64     103.941276     30.754493   620.905375       NaN    41.285580  ...   59.448387
   3     3           1       104.062539       30.739077       311     104.060024     30.742693   467.552293       NaN    37.796143  ...   46.950858
   4     4           1       104.062539       30.739077      1288     104.062071     30.732501   730.287581       NaN    60.548947  ...   80.810511
 ...   ...         ...              ...             ...       ...            ...           ...          ...       ...          ...  ...         ...
5938  5938        1900       104.029570       30.670992      1729     104.031714     30.670676   208.333343       NaN    15.136227  ...   21.428572
5939  5939        1901       104.025686       30.626472      1233     104.023241     30.622808   468.731933       NaN    40.644760  ...   43.083732
5940  5940        1901       104.025686       30.626472      1565     104.023113     30.627839   289.473210       NaN    50.246073  ...   39.523523
5941  5941        1901       104.025686       30.626472      1658     104.028963     30.624882   360.123488       NaN    36.935304  ...   40.066719
```



```python
In [57]:  av_time = wday_off.pop('avg_time')
          wday_off.insert(8, 'Time', av_time)

          av_time = wday_peak.pop('avg_time')
          wday_peak.insert(8, 'Time', av_time)

          av_time = wend_off.pop('avg_time')
          wend_off.insert(8, 'Time', av_time)


          av_time = wend_peak.pop('avg_time')
          wend_peak.insert(8, 'Time', av_time)
```

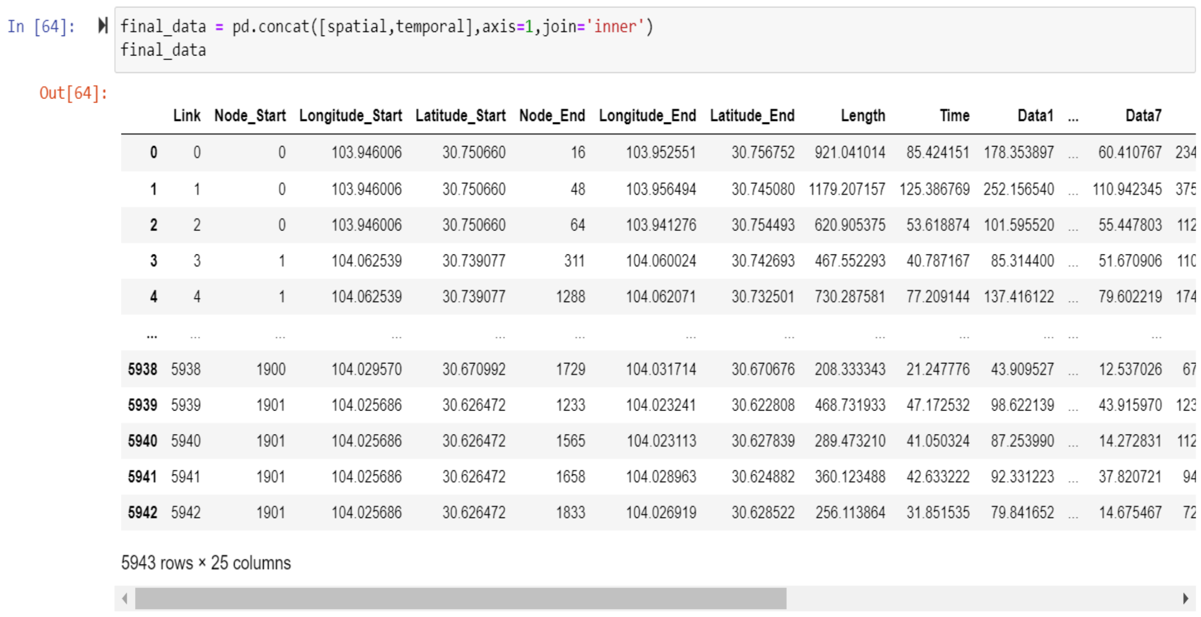


```python
In [58]:  wday_off_spatial = wday_off.iloc[:,:9]
          wday_peak_spatial = wday_peak.iloc[:,:9]
          wend_off_spatial = wend_off.iloc[:,:9]
          wend_peak_spatial = wend_peak.iloc[:,:9]
```



```
In [58]:  wday_off_spatial

Out[58]:
      Link  Node_Start  Longitude_Start  Latitude_Start  Node_End  Longitude_End  Latitude_End       Length        Time
   0     0           0       103.946006       30.750660        16     103.952551     30.756752   921.041014   85.424151
   1     1           0       103.946006       30.750660        48     103.956494     30.745080  1179.207157  125.386769
   2     2           0       103.946006       30.750660        64     103.941276     30.754493   620.905375   53.618874
   3     3           1       104.062539       30.739077       311     104.060024     30.742693   467.552294   40.787167
   4     4           1       104.062539       30.739077      1288     104.062071     30.732501   730.287581   77.209144
 ...   ...         ...              ...             ...       ...            ...           ...          ...         ...
5938  5938        1900       104.029570       30.670992      1729     104.031714     30.670676   208.333343   21.247776
5939  5939        1901       104.025686       30.626472      1233     104.023241     30.622808   468.731933   47.172532
5940  5940        1901       104.025686       30.626472      1565     104.023113     30.627839   289.473210   41.050324
5941  5941        1901       104.025686       30.626472      1658     104.028963     30.624882   360.123488   42.633222
5942  5942        1901       104.025686       30.626472      1833     104.026919     30.628522   256.113864   31.851535

5943 rows × 9 columns
```

For our final data, we merge this spatial data with the temporal data received after the application of autoencoder.


```
In [64]:  final_data = pd.concat([spatial,temporal],axis=1,join='inner')
          final_data
```

Out[64]:

	Link	Node_Start	Longitude_Start	Latitude_Start	Node_End	Longitude_End	Latitude_End	Length	Time	Data1	...	Data7	
0	0	0	103.946006	30.750660	16	103.952551	30.756752	921.041014	85.424151	178.353897	...	60.410767	234
1	1	0	103.946006	30.750660	48	103.956494	30.745080	1179.207157	125.386769	252.156540	...	110.942345	375
2	2	0	103.946006	30.750660	64	103.941276	30.754493	620.905375	53.618874	101.595520	...	55.447803	112
3	3	1	104.062539	30.739077	311	104.060024	30.742693	467.552293	40.787167	85.314400	...	51.670906	110
4	4	1	104.062539	30.739077	1288	104.062071	30.732501	730.287581	77.209144	137.416122	...	79.602219	174
...	...	...	...	...	...	...	...	...	...	...	...	...	
5938	5938	1900	104.029570	30.670992	1729	104.031714	30.670676	208.333343	21.247776	43.909527	...	12.537026	67
5939	5939	1901	104.025686	30.626472	1233	104.023241	30.622808	468.731933	47.172532	98.622139	...	43.915970	123
5940	5940	1901	104.025686	30.626472	1565	104.023113	30.627839	289.473210	41.050324	87.253990	...	14.272831	112
5941	5941	1901	104.025686	30.626472	1658	104.028963	30.624882	360.123488	42.633222	92.331223	...	37.820721	94
5942	5942	1901	104.025686	30.626472	1833	104.026919	30.628522	256.113864	31.851535	79.841652	...	14.675467	72

5943 rows × 25 columns

Graph Neural Network [GNN]

We have already discussed what GNN is and why it is best to use it specially when dealing with non-Euclidian spaces. Now, let us see the implementation in detail.

A graph is a data structure comprising of nodes (vertices) and edges connected together to represent information with no definite beginning or end. All the nodes occupy an arbitrary position in space, usually clustered according to similar features when plotted in *2D* (or even *nD*) space [we will plot our graph in the upcoming steps]. Each node has a set of features defining it. In the case of Time Travel Estimation, this could be age, the position (latitude and longitude) corresponding to that particular node. Each edge connects two nodes together and shows some kind of interaction or relationship between them.

Once the conversion of nodes and edges are completed, the graph performs Message Passing between the nodes. This process is also called Neighbourhood Aggregation because it involves pushing messages (aka, the embeddings) from surrounding nodes around a given reference node, through the directed edges. In terms of GNNs, for a single reference node, the neighbouring nodes pass their messages (embeddings) through the edge neural networks into the recurrent unit on the reference node. The new embedding of the reference recurrent unit is updated by applying said recurrent function on the current embedding and a summation of the edge neural network outputs of the neighbouring node embeddings. This process is performed, in parallel, on all nodes in the network as embeddings in layer *L+1* depend on embeddings in layer *L*. Which is why, in practice, we don’t need to ‘move’ from one node to another to carry out Message Passing.

How is this useful in Travel Time Estimation?

We treat the local road network as a graph, where each route segment corresponds to a node and edges exist between segments that are consecutive on the same road or connected through an intersection.

In a Graph Neural Network, a message passing algorithm is executed where the messages and their effect on edge and node states are learned by neural networks. From this viewpoint, our Super segments are road subgraphs, which were sampled at random in proportion to traffic density. A single model can therefore be trained using these sampled subgraphs, and can be deployed at scale. As an example of the same, we will perform clustering in the upcoming section.

Graph Neural Networks extend the learning bias imposed by Convolutional Neural Networks and Recurrent Neural Networks by generalising the concept of “proximity”, allowing us to have arbitrarily complex connections to handle not only traffic ahead or behind us, but also along adjacent and intersecting roads.

As discussed, in a Graph Neural Network, adjacent nodes pass messages to each other. By keeping this structure, we impose a locality bias where nodes will find it easier to rely on adjacent nodes (this only requires one message passing step). These mechanisms allow Graph Neural Networks to capitalise on the connectivity structure of the road network more effectively. Think of how a jam on a side street can spill over to affect traffic on a larger road. By spanning multiple intersections, the model gains the ability to natively predict delays at turns, delays due to merging, and the overall traversal time in stop-and-go traffic. This ability of Graph Neural Networks to generalise over combinatorial spaces is what grants our modelling technique its power. Each Super segment, which can be of varying length and of varying complexity - from simple two-segment routes to longer routes containing hundreds of nodes - can nonetheless be processed by the **same** Graph Neural Network model.

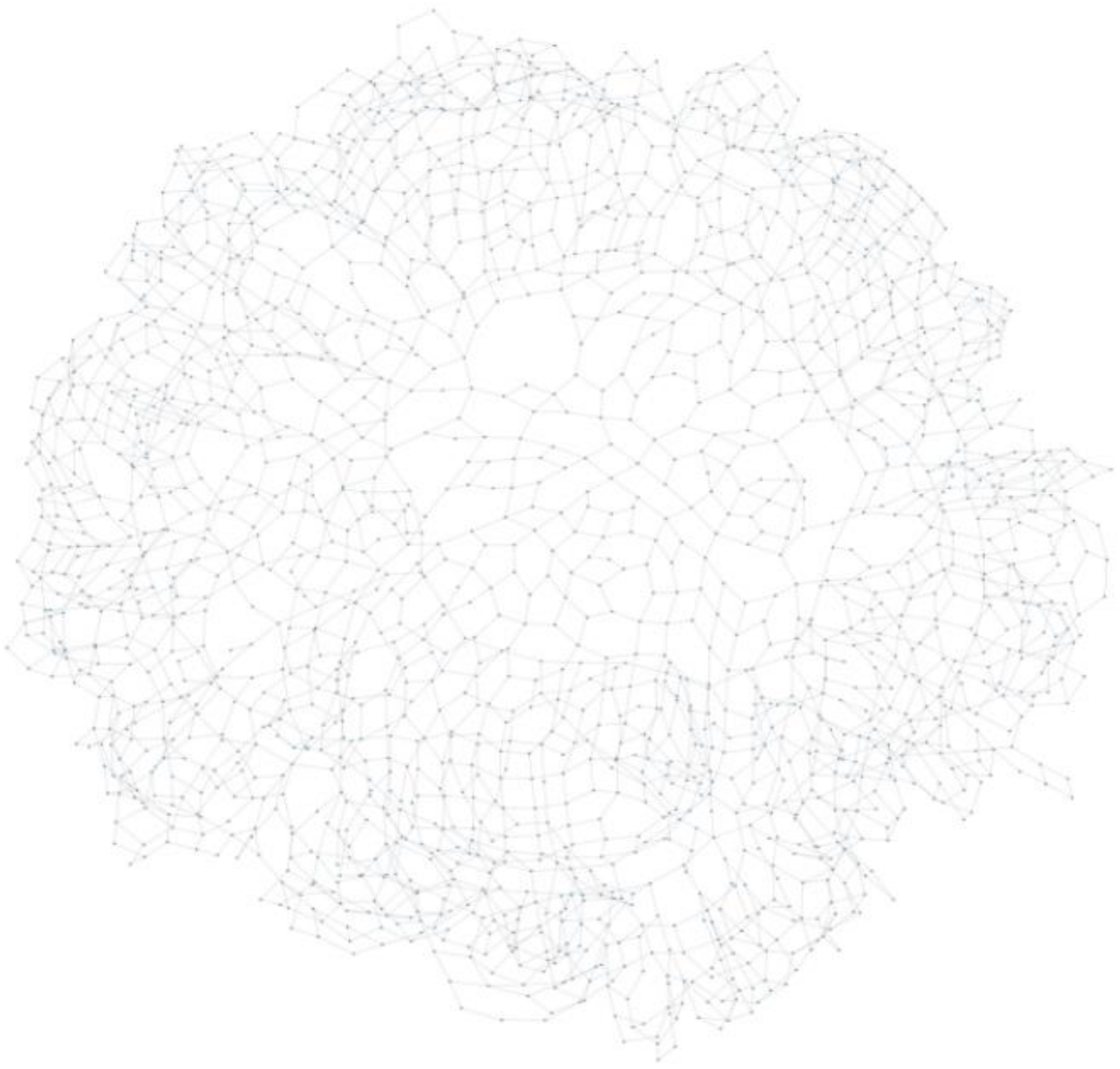
Defining node start and node end-

In [73]:
```
node_start['Link'] = node_start['Link'].astype('int64')
node_start['Node_Start'] = node_start['Node_Start'].astype('int64')
node_start['Node_End'] = node_start['Node_End'].astype('int64')
```

Generating a graph for our spatial data

In [85]:
```
import dgl
src = spatial['Node_Start'].to_numpy()
dst = spatial['Node_End'].to_numpy()
# Create a DGL graph from a pair of numpy arrays
g = dgl.graph((src, dst))
```

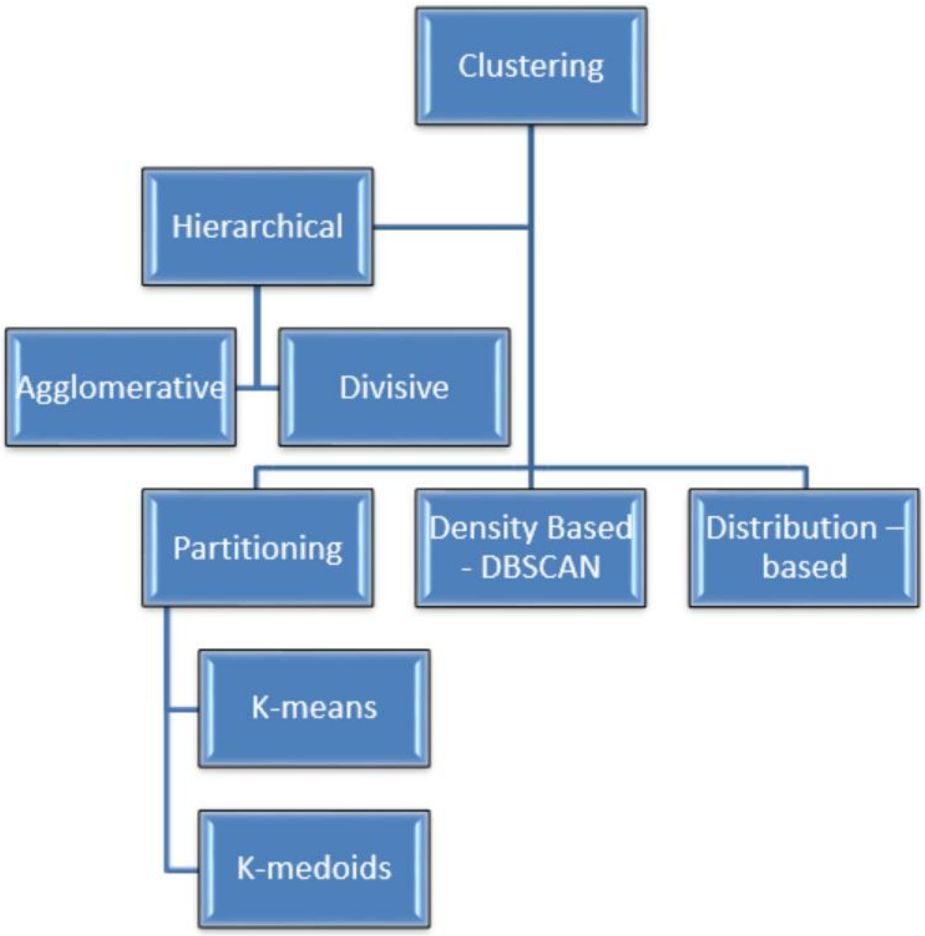
In [86]:
```
import networkx as nx
# Since the actual graph is undirected, we convert it for visualization purpose.
nx_g = g.to_networkx().to_undirected()
plt.figure(figsize=[150,150])
pos = nx.kamada_kawai_layout(nx_g)
nx.draw(nx_g,pos, with_labels=True)
```



Clustering

Clustering is the task of dividing the population or data points into a number of groups such that data points in the same groups are more similar to other data points in the same group and dissimilar to the data points in other groups. It is basically a collection of objects on the basis of similarity and dissimilarity between them. It is basically a type of unsupervised learning method. We have seen how grouping or sampling of our graph data can be useful in traffic time estimation. There are no criteria for good clustering. It depends on the user, what is the criteria they may use which satisfy their need. The sampling can be done in accordance with the traffic density, or topology or some other parameter based on the requirement. By being able to group a particular set of nodes, carrying out tasks, such as extracting a particular set of nodes will take less time.

Clustering methods- Based on the type of clustering algorithm used, there are different types of clustering methods:



We will be implementing Hierarchical Clustering for our model.

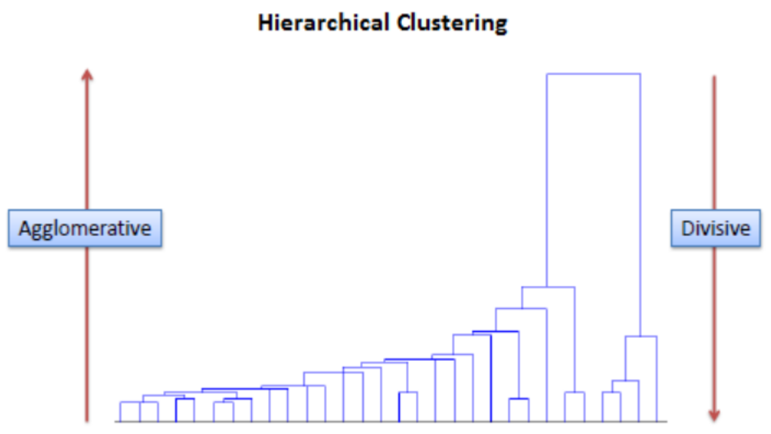
Hierarchical clustering: It is a tree-based clustering method where the observations are divided into a tree like structure using distance as a measure. It is a method of cluster analysis which seeks to build a hierarchy of clusters.

Why Hierarchical clustering?

For our application, distance is the measure of similarity and hence observations which have smaller distance are considered as similar and vice-versa. This type of clustering technique is also known as **connectivity-based** method. In this method, simple partitioning of the data set will not be done, whereas it provides us with the hierarchy of the clusters that merge after a certain distance. After the hierarchical clustering is done on the dataset, the result will be a tree-based representation of data points [Dendrogram], divided into clusters. One of the biggest advantages of hierarchical clustering is that, **unlike k-means clustering**, we do not have to specify the number of clusters as input to the algorithm. Also, with the Ease of handling of any forms of similarity or distance provided, it seems only fair to use Hierarchical Clustering for our application.

The distance measured can be Euclidian, Manhattan or Minkowski. For our application, we use Euclidian distance which is simply calculated using Pythagoras theorem.

Hierarchical clustering is further classified into two types — Agglomerative Clustering and Divisive Clustering.

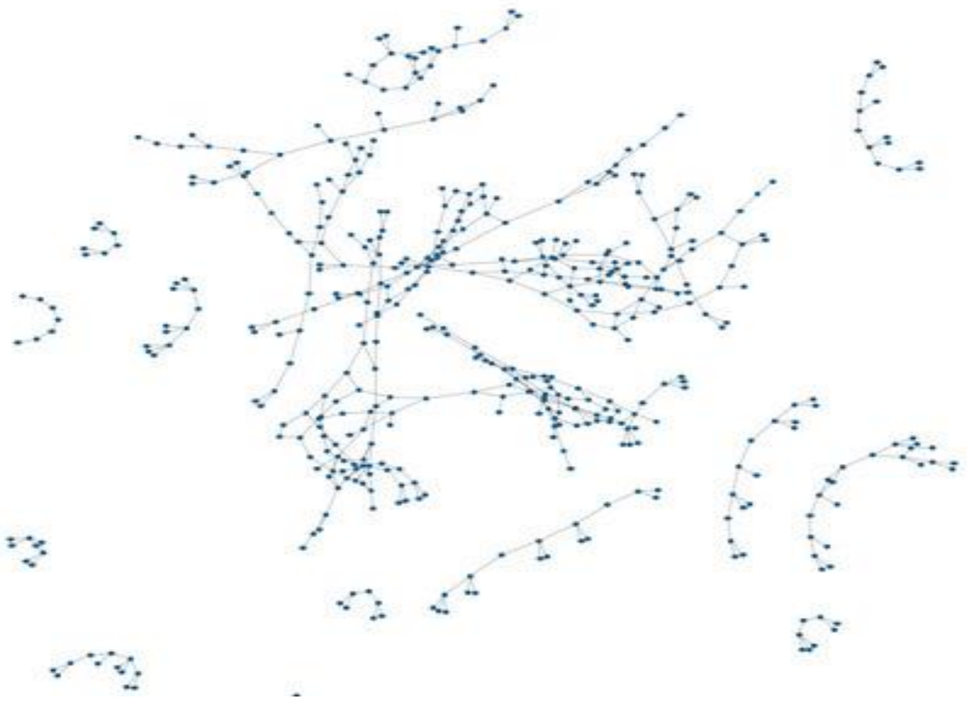


Because we are trying to divide our map, we use divisive clustering instead of agglomerative clustering.

Divisive clustering -This type of clustering uses a top-down approach in which all observations are initially grouped into one single cluster. The largest clustering is then split into two clusters in such a way that the average distance between the two clusters is maximum. This step is recursively performed until every individual observation forms a separate cluster.

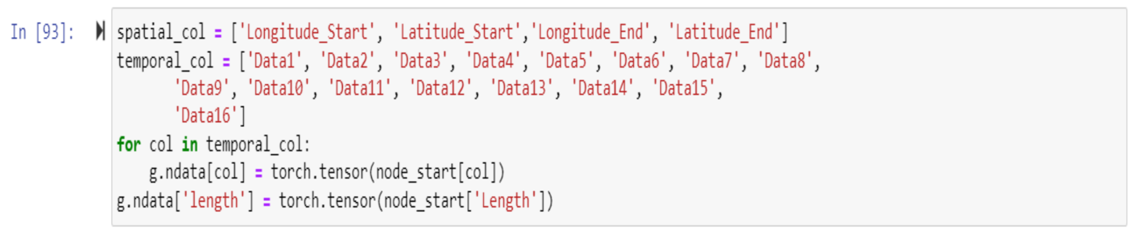
The mostly used algorithm has a time complexity of O(n^3) and requires Ω(n^2) memory. This shows that the algorithm is too slow even if we take a medium scale dataset into consideration. However, we can possibly tweak the algorithm so that we can reduce the time complexity of the clustering to O(n^2.log n), This time could be taken down even lower by using faster heuristics to choose splits, such as k-means, etc. Still, the dataset that we will be using, the time complexity can be said as quite big. But considering the fact that topology of the area would not change drastically, we can run clustering once and then we consider the clusters as object and could be used for preparing the model. Although, if new routes are added in those areas, we might need to re-run the clustering segment.

Clustered Data

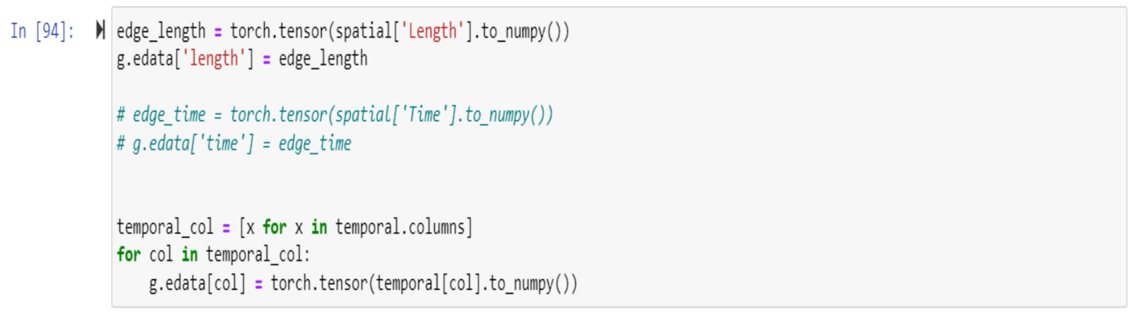


Final GNN Model-

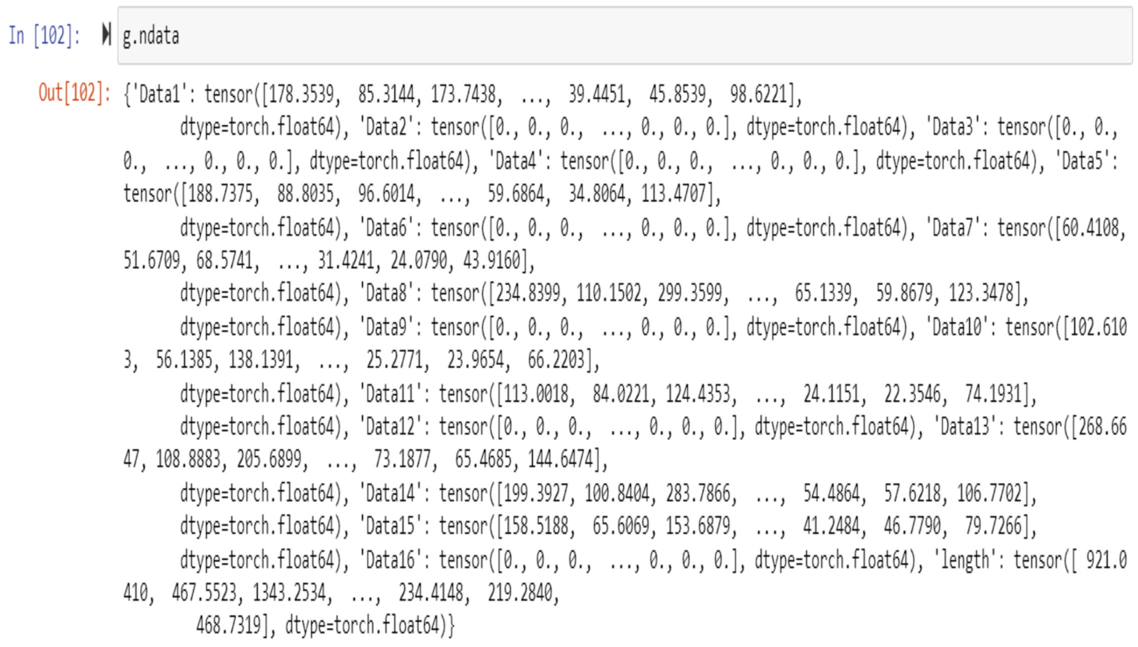
Preparing the input data-



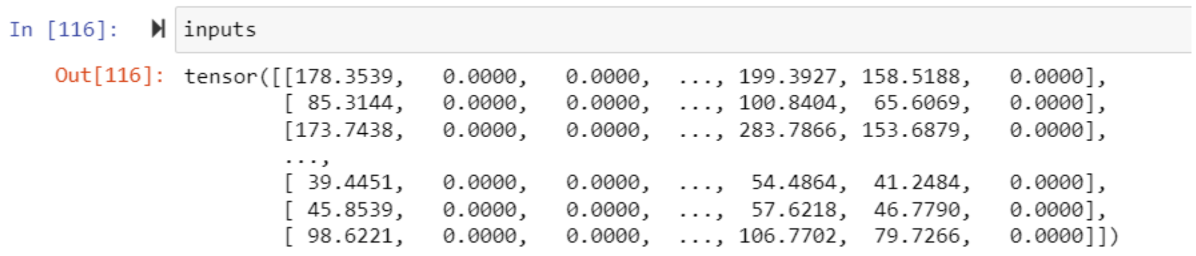
```
In [93]:  spatial_col = ['Longitude_Start', 'Latitude_Start','Longitude_End', 'Latitude_End']
          temporal_col = ['Data1', 'Data2', 'Data3', 'Data4', 'Data5', 'Data6', 'Data7', 'Data8',
                 'Data9', 'Data10', 'Data11', 'Data12', 'Data13', 'Data14', 'Data15',
                 'Data16']
          for col in temporal_col:
              g.ndata[col] = torch.tensor(node_start[col])
          g.ndata['length'] = torch.tensor(node_start['Length'])
```



```
In [94]:  edge_length = torch.tensor(spatial['Length'].to_numpy())
          g.edata['length'] = edge_length

          # edge_time = torch.tensor(spatial['Time'].to_numpy())
          # g.edata['time'] = edge_time


          temporal_col = [x for x in temporal.columns]
          for col in temporal_col:
              g.edata[col] = torch.tensor(temporal[col].to_numpy())
```

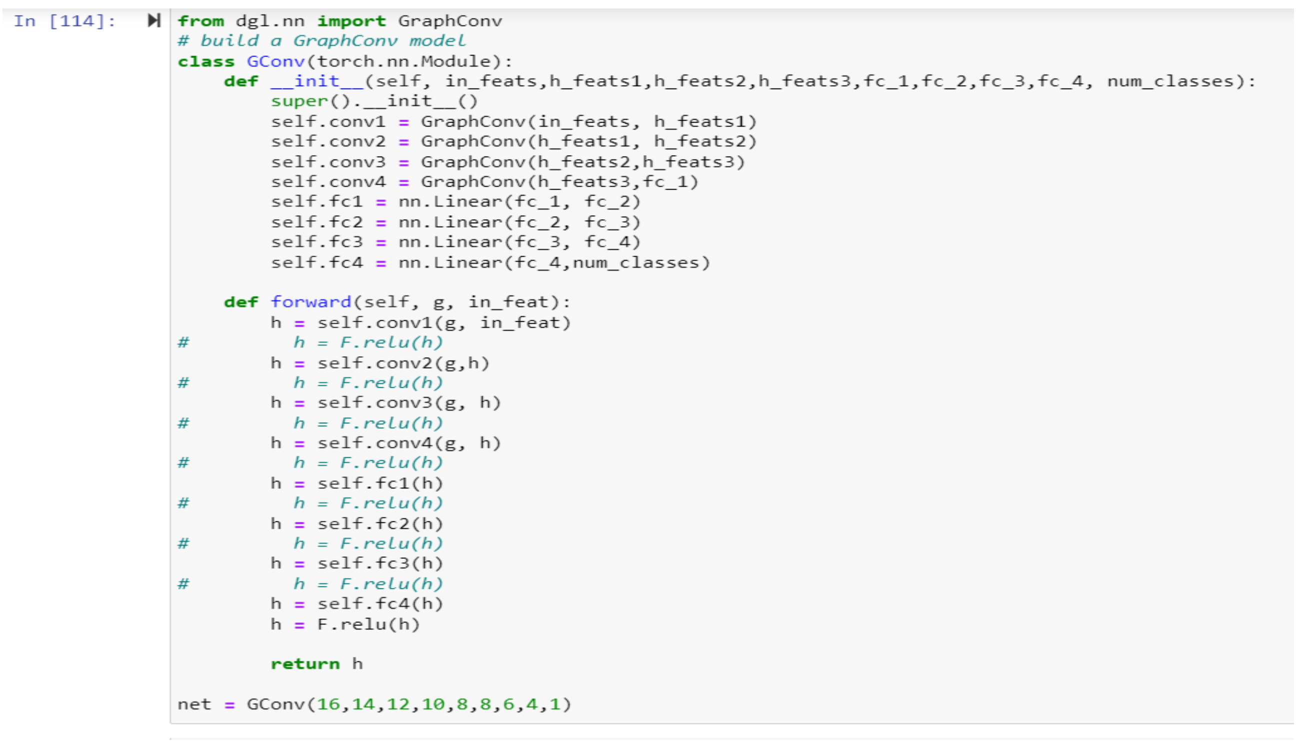


```
In [102]:  g.ndata

Out[102]: {'Data1': tensor([178.3539,  85.3144, 173.7438,  ...,  39.4451,  45.8539,  98.6221],
                 dtype=torch.float64), 'Data2': tensor([0., 0., 0.,  ..., 0., 0., 0.], dtype=torch.float64), 'Data3': tensor([0., 0.,
          0.,  ..., 0., 0., 0.], dtype=torch.float64), 'Data4': tensor([0., 0., 0.,  ..., 0., 0., 0.], dtype=torch.float64), 'Data5':
          tensor([188.7375,  88.8035,  96.6014,  ...,  59.6864,  34.8064, 113.4707],
                 dtype=torch.float64), 'Data6': tensor([0., 0., 0.,  ..., 0., 0., 0.], dtype=torch.float64), 'Data7': tensor([60.4108,
          51.6709, 68.5741,  ..., 31.4241, 24.0790, 43.9160],
                 dtype=torch.float64), 'Data8': tensor([234.8399, 110.1502, 299.3599,  ...,  65.1339,  59.8679, 123.3478],
                 dtype=torch.float64), 'Data9': tensor([0., 0., 0.,  ..., 0., 0., 0.], dtype=torch.float64), 'Data10': tensor([102.610
          3,  56.1385, 138.1391,  ...,  25.2771,  23.9654,  66.2203],
                 dtype=torch.float64), 'Data11': tensor([113.0018,  84.0221, 124.4353,  ...,  24.1151,  22.3546,  74.1931],
                 dtype=torch.float64), 'Data12': tensor([0., 0., 0.,  ..., 0., 0., 0.], dtype=torch.float64), 'Data13': tensor([268.66
          47, 108.8883, 205.6899,  ...,  73.1877,  65.4685, 144.6474],
                 dtype=torch.float64), 'Data14': tensor([199.3927, 100.8404, 283.7866,  ...,  54.4864,  57.6218, 106.7702],
                 dtype=torch.float64), 'Data15': tensor([158.5188,  65.6069, 153.6879,  ...,  41.2484,  46.7790,  79.7266],
                 dtype=torch.float64), 'Data16': tensor([0., 0., 0.,  ..., 0., 0., 0.], dtype=torch.float64), 'length': tensor([ 921.0
          410,  467.5523, 1343.2534,  ...,  234.4148,  219.2840,
                   468.7319], dtype=torch.float64)}
```



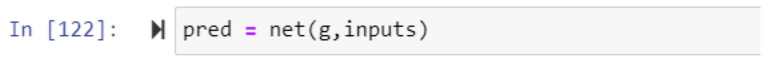
```
In [116]:  inputs

Out[116]:  tensor([[178.3539,   0.0000,   0.0000,  ..., 199.3927, 158.5188,   0.0000],
                   [ 85.3144,   0.0000,   0.0000,  ..., 100.8404,  65.6069,   0.0000],
                   [173.7438,   0.0000,   0.0000,  ..., 283.7866, 153.6879,   0.0000],
                   ...,
                   [ 39.4451,   0.0000,   0.0000,  ...,  54.4864,  41.2484,   0.0000],
                   [ 45.8539,   0.0000,   0.0000,  ...,  57.6218,  46.7790,   0.0000],
                   [ 98.6221,   0.0000,   0.0000,  ..., 106.7702,  79.7266,   0.0000]])
```

Defining the GNN model-



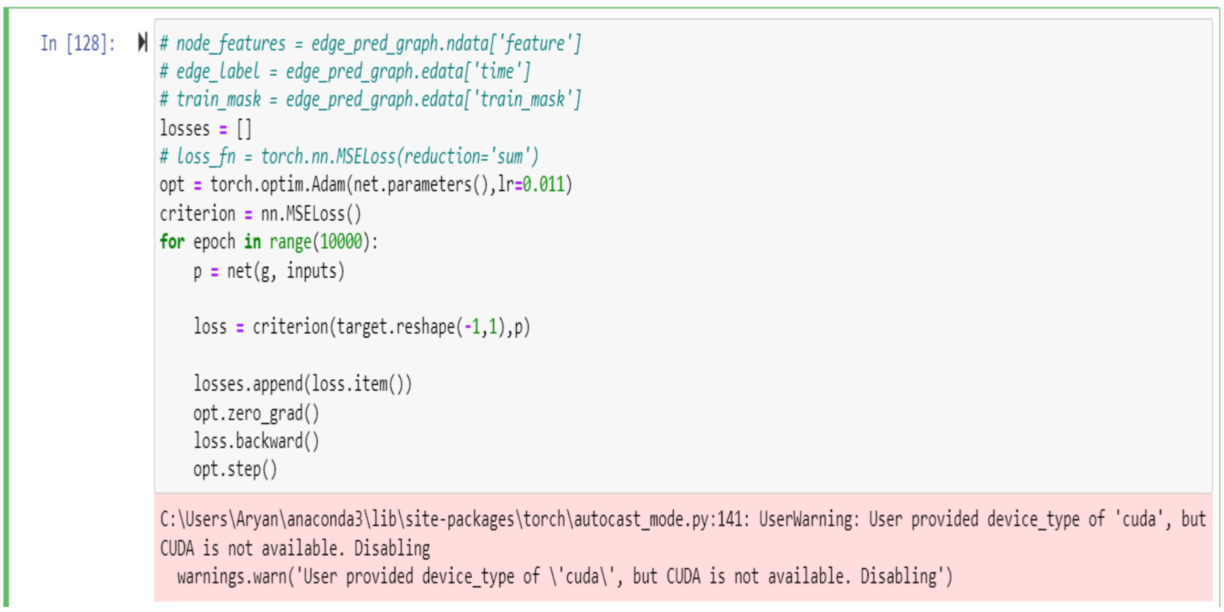
```
In [114]:  from dgl.nn import GraphConv
           # build a GraphConv model
           class GConv(torch.nn.Module):
               def __init__(self, in_feats,h_feats1,h_feats2,h_feats3,fc_1,fc_2,fc_3,fc_4, num_classes):
                   super().__init__()
                   self.conv1 = GraphConv(in_feats, h_feats1)
                   self.conv2 = GraphConv(h_feats1, h_feats2)
                   self.conv3 = GraphConv(h_feats2,h_feats3)
                   self.conv4 = GraphConv(h_feats3,fc_1)
                   self.fc1 = nn.Linear(fc_1, fc_2)
                   self.fc2 = nn.Linear(fc_2, fc_3)
                   self.fc3 = nn.Linear(fc_3, fc_4)
                   self.fc4 = nn.Linear(fc_4,num_classes)

               def forward(self, g, in_feat):
                   h = self.conv1(g, in_feat)
           #         h = F.relu(h)
                   h = self.conv2(g,h)
           #         h = F.relu(h)
                   h = self.conv3(g, h)
           #         h = F.relu(h)
                   h = self.conv4(g, h)
           #         h = F.relu(h)
                   h = self.fc1(h)
           #         h = F.relu(h)
                   h = self.fc2(h)
           #         h = F.relu(h)
                   h = self.fc3(h)
           #         h = F.relu(h)
                   h = self.fc4(h)
                   h = F.relu(h)

                   return h

           net = GConv(16,14,12,10,8,8,6,4,1)
```

Feeding the input to our model-



```
In [122]:  pred = net(g,inputs)
```

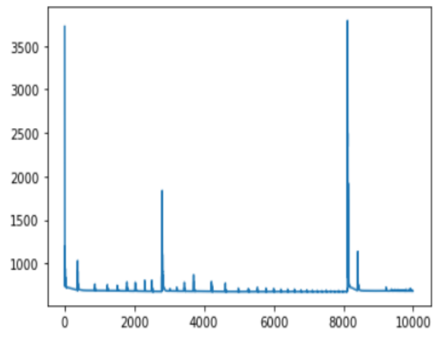
Defining Losses-



```
In [128]:  # node_features = edge_pred_graph.ndata['feature']
           # edge_label = edge_pred_graph.edata['time']
           # train_mask = edge_pred_graph.edata['train_mask']
           losses = []
           # loss_fn = torch.nn.MSELoss(reduction='sum')
           opt = torch.optim.Adam(net.parameters(),lr=0.011)
           criterion = nn.MSELoss()
           for epoch in range(10000):
               p = net(g, inputs)

               loss = criterion(target.reshape(-1,1),p)

               losses.append(loss.item())
               opt.zero_grad()
               loss.backward()
               opt.step()

           C:\Users\Aryan\anaconda3\lib\site-packages\torch\autocast_mode.py:141: UserWarning: User provided device_type of 'cuda', but
           CUDA is not available. Disabling
             warnings.warn('User provided device_type of \'cuda\', but CUDA is not available. Disabling')
```

Loss graph with number of epochs-

```
In [129]:    losses = [x for x in losses if x<3793]

In [130]:    plt.plot(losses)

Out[130]: [<matplotlib.lines.Line2D at 0x28c376e44c0>]
```



Final Predicted Output-

```
In [131]:    p

Out[131]: tensor([[124.9273],
                  [ 77.8484],
                  [190.1577],
                  ...,
                  [ 41.7466],
                  [ 35.7206],
                  [ 42.9183]], grad_fn=<ReluBackward0>)
```

Conclusion

The predicted output is similar to the actual output, indicating minimal loss.

We have successfully implemented our Deep-Learning based model on Travel Time Estimation.