```
import pandas as pd
import glob
import os

# setting the path for joining multiple files
files = os.path.join("/content/drive/MyDrive/data/s3Files", "data*.txt")

# list of merged files returned
files = glob.glob(files)


df = pd.DataFrame()


for file in files:

    d = pd.read_csv(file,header=None)
    df = df.append(d[[1,2,3]], ignore_index=True)

df.head()
```

	1	2	3
0	2102	2318	97
1	2095	2326	94
2	2087	2336	78
3	1867	2429	283
4	1862	2430	297

```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
mms.fit(df)
```

```
▾ MinMaxScaler
MinMaxScaler()
```

```
df = mms.transform(df)


df = pd.DataFrame(df)
df.head()
```

	0	1	2
0	0.689034	0.717666	0.106946
1	0.677578	0.730284	0.103638
2	0.664484	0.746057	0.085998
3	0.304419	0.892744	0.312018
4	0.296236	0.894322	0.327453

```
train = df[:2536000]
test = df[2536000:]


train.shape
```

```
(2536000, 3)
```

```
test.shape
```

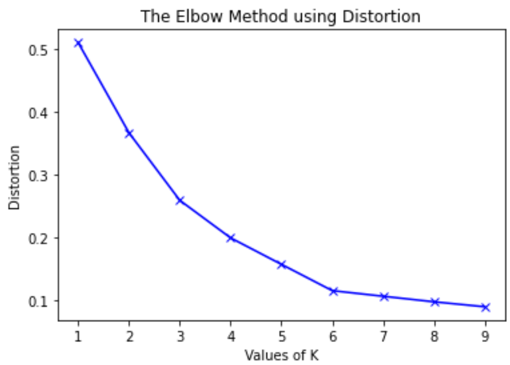
```
(634000, 3)
```

```
from sklearn.cluster import KMeans
from sklearn import metrics
from scipy.spatial.distance import cdist
import numpy as np
import matplotlib.pyplot as plt

distortions = []
inertias = []
mapping1 = {}
```

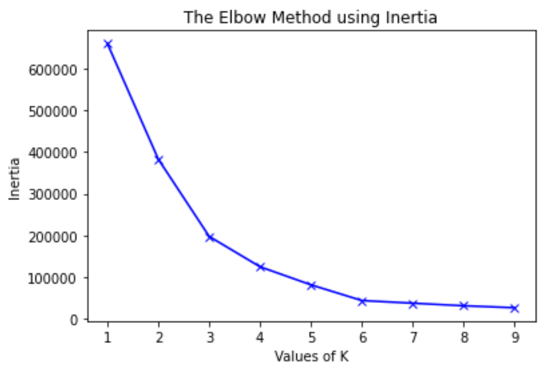
```
mapping2 = {}
K = range(1, 10)

for k in K:
    # Building and fitting the model
    kmeanModel = KMeans(n_clusters=k).fit(train)

    distortions.append(sum(np.min(cdist(train, kmeanModel.cluster_centers_,
                                        'euclidean'), axis=1)) / train.shape[0])
    inertias.append(kmeanModel.inertia_)

    mapping1[k] = sum(np.min(cdist(train, kmeanModel.cluster_centers_,
                                   'euclidean'), axis=1)) / train.shape[0]
    mapping2[k] = kmeanModel.inertia_
```

```
/usr/local/lib/python3.8/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
```

```
plt.plot(K, distortions, 'bx-')
plt.xlabel('Values of K')
plt.ylabel('Distortion')
plt.title('The Elbow Method using Distortion')
plt.show()
```




```
plt.plot(K, inertias, 'bx-')
plt.xlabel('Values of K')
plt.ylabel('Inertia')
plt.title('The Elbow Method using Inertia')
plt.show()
```




```
train.head()
```

	0	1	2
0	0.689034	0.717666	0.106946
1	0.677578	0.730284	0.103638
2	0.664484	0.746057	0.085998
3	0.304419	0.892744	0.312018

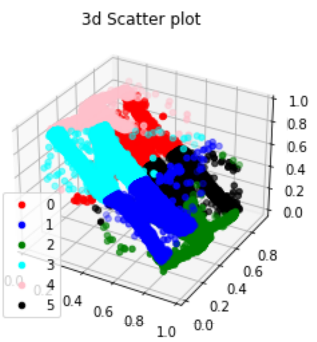
```
kmeanModel = KMeans(n_clusters=6).fit(train)
train[3] = kmeanModel.labels_
```

```
/usr/local/lib/python3.8/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
<ipython-input-16-ab78c8f6464a>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus
  train[3] = kmeanModel.labels_
```

```
train.head()
```

	0	1	2	3
0	0.689034	0.717666	0.106946	5
1	0.677578	0.730284	0.103638	5
2	0.664484	0.746057	0.085998	5
3	0.304419	0.892744	0.312018	0
4	0.296236	0.894322	0.327453	0

```
# importing mplot3d toolkits
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()

# syntax for 3-D projection
ax = plt.axes(projection ='3d')

# defining axes
for i,c in zip([0,1,2,3,4,5],['red','blue','green','cyan','pink','black']):
  t = train[train[3]==i]
  z = t[2]
  x = t[0]
  y = t[1]

  ax.scatter(x, y, z, c=c )

# syntax for plotting
ax.set_title('3d Scatter plot')
ax.legend(['0','1','2','3','4','5'])
plt.show()
```

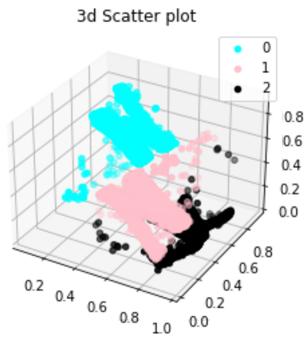



```
kmeanModel = KMeans(n_clusters=3).fit(train)
train[4] = kmeanModel.labels_
```

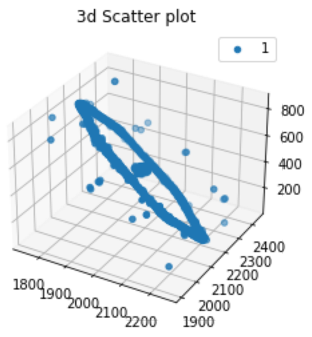
```
/usr/local/lib/python3.8/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change fro
  warnings.warn(
<ipython-input-25-216335e8719f>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
```

```
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus
  train[4] = kmeanModel.labels_
```

```
fig = plt.figure()

# syntax for 3-D projection
ax = plt.axes(projection ='3d')

# defining axes
for i,c in zip([0,1,2],['cyan','pink','black']):
  t = train[train[3]==i]
  z = t[2]
  x = t[0]
  y = t[1]

  ax.scatter(x, y, z, c=c, )

# syntax for plotting
ax.set_title('3d Scatter plot')
ax.legend(['0','1','2'])
plt.show()
```

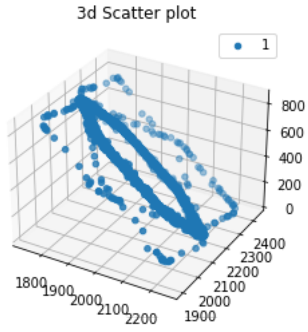



```
fig = plt.figure()

# syntax for 3-D projection
ax = plt.axes(projection ='3d')
z = d[3]
x = d[1]
y = d[2]

ax.scatter(x, y, z)

# syntax for plotting
ax.set_title('3d Scatter plot')
ax.legend(['1','2','3'])
plt.show()
```

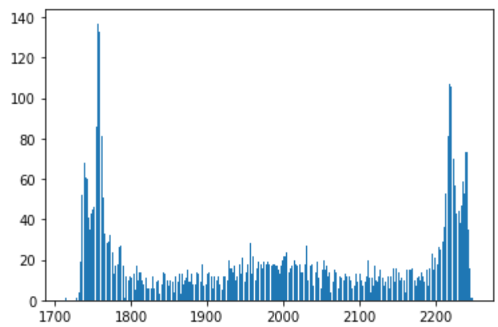



```
d = pd.read_csv("/content/drive/MyDrive/data/s3Files/data5.txt",header=None)
d = d[[1,2,3]]


fig = plt.figure()

# syntax for 3-D projection
ax = plt.axes(projection ='3d')
z = d[3]
x = d[1]
y = d[2]
```

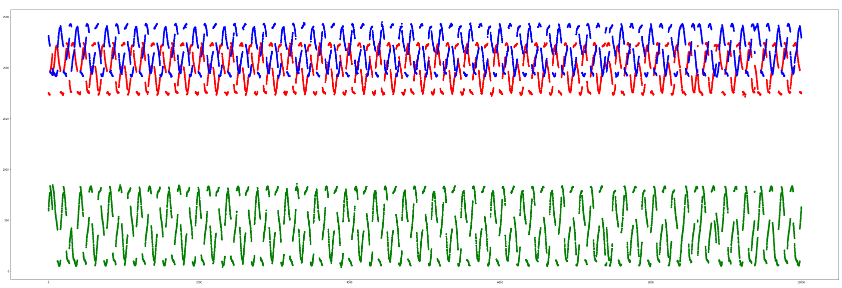
```
ax.scatter(x, y, z)

# syntax for plotting
ax.set_title('3d Scatter plot')
ax.legend(['1','2','3'])
plt.show()
```



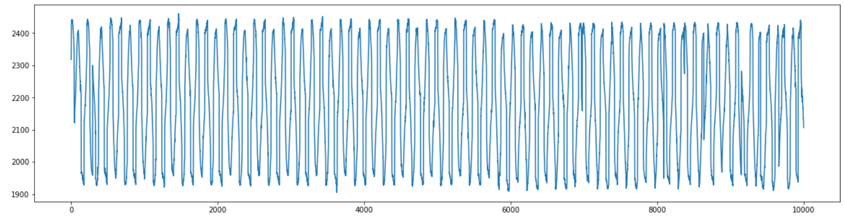

```
i = d[1].value_counts()

plt.bar(i.index,i.values)
plt.show()
```



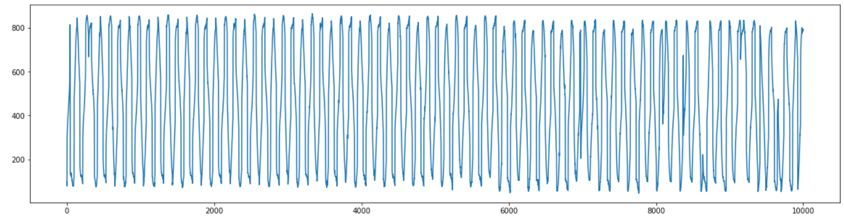

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(60,20))
#plt.plot(range(len(d[1])),d[1],color='red')
plt.scatter(range(len(d[1])),d[1],color='red')
#plt.plot(range(len(d[1])),d[2],color='blue')
plt.scatter(range(len(d[1])),d[2],color='blue')
#plt.plot(range(len(d[1])),d[3],color='green')
plt.scatter(range(len(d[1])),d[3],color='green')
plt.show()
```



```
fig = plt.figure(figsize=(20,5))
plt.plot(range(len(d[2])),d[2])
plt.show()
```




```
fig = plt.figure(figsize=(20,5))
plt.plot(range(len(d[3])),d[3])
plt.show()
```




```
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Model
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler


data = np.array(df).reshape(12680,250,3)


flatten_layer = tf.keras.layers.Flatten()
Data = flatten_layer(data)


Data.shape

     TensorShape([12680, 750])


learning_rate = 0.01
training_epochs = 50
batch_size = 256
display_step = 1
examples_to_show = 10
global_step = tf.Variable(0)
total_batch = int(len(data) / batch_size)

# Network Parameters
n_hidden_1 = 256 # 1st layer num features
n_hidden_2 = 128 # 2nd layer num features
encoding_layer = 32 # final encoding bottleneck features
n_input = 750 # MNIST data input (img shape: 28*28)


enocoding_1 = tf.keras.layers.Dense(n_hidden_1, activation=tf.nn.sigmoid)
encoding_2 = tf.keras.layers.Dense(n_hidden_2, activation=tf.nn.sigmoid)
encoding_final = tf.keras.layers.Dense(encoding_layer, activation=tf.nn.relu)

# Building the encoder
def encoder(x):
    x_reshaped = flatten_layer(x)
    # Encoder first layer with sigmoid activation #1
    layer_1 = enocoding_1(x_reshaped)
```

```
    # Encoder second layer with sigmoid activation #2
    layer_2 = encoding_2(layer_1)
    code = encoding_final(layer_2)
    return code

decoding_1 = tf.keras.layers.Dense(n_hidden_2, activation=tf.nn.sigmoid)
decoding_2 = tf.keras.layers.Dense(n_hidden_1, activation=tf.nn.sigmoid)
decoding_final = tf.keras.layers.Dense(n_input)
# Building the decoder
def decoder(x):
    # Decoder first layer with sigmoid activation #1
    layer_1 = decoding_1(x)
    # Decoder second layer with sigmoid activation #2
    layer_2 = decoding_2(layer_1)
    decode = decoding_final(layer_2)
    return decode


class AutoEncoder(tf.keras.Model):
    def __init__(self):
        super(AutoEncoder, self).__init__()

        self.n_hidden_1 = n_hidden_1 # 1st layer num features
        self.n_hidden_2 = n_hidden_2 # 2nd layer num features
        self.encoding_layer = encoding_layer
        self.n_input = n_input

        self.flatten_layer = tf.keras.layers.Flatten()
        self.enocoding_1 = tf.keras.layers.Dense(self.n_hidden_1, activation=tf.nn.sigmoid)
        self.encoding_2 = tf.keras.layers.Dense(self.n_hidden_2, activation=tf.nn.sigmoid)
        self.encoding_final = tf.keras.layers.Dense(self.encoding_layer, activation=tf.nn.relu)
        self.decoding_1 = tf.keras.layers.Dense(self.n_hidden_2, activation=tf.nn.sigmoid)
        self.decoding_2 = tf.keras.layers.Dense(self.n_hidden_1, activation=tf.nn.sigmoid)
        self.decoding_final = tf.keras.layers.Dense(self.n_input)


    # Building the encoder
    def encoder(self,x):
        #x = self.flatten_layer(x)
        layer_1 = self.enocoding_1(x)
        layer_2 = self.encoding_2(layer_1)
        code = self.encoding_final(layer_2)
        return code


    # Building the decoder
    def decoder(self, x):
        layer_1 = self.decoding_1(x)
        layer_2 = self.decoding_2(layer_1)
        decode = self.decoding_final(layer_2)
        return decode


    def call(self, x):
        encoder_op  = self.encoder(x)
        # Reconstructed Images
        y_pred = self.decoder(encoder_op)
        return y_pred

def cost(y_true, y_pred):
    loss = tf.losses.mean_squared_error(y_true, y_pred)
    cost = tf.reduce_mean(loss)
    return cost

def grad(model, inputs, targets):
    #print('shape of inputs : ',inputs.shape)
    #targets = flatten_layer(targets)
    with tf.GradientTape() as tape:
        reconstruction = model(inputs)
        loss_value = cost(targets, reconstruction)
    return loss_value, tape.gradient(loss_value, model.trainable_variables),reconstruction


model = AutoEncoder()
optimizer = tf.keras.optimizers.RMSprop(learning_rate)

for epoch in range(training_epochs):
    for i in range(total_batch):
        x_inp = Data[i : i + batch_size]
        loss_value, grads, reconstruction = grad(model, x_inp, x_inp)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
    # Display logs per epoch step
    if epoch % display_step == 0:
```

```
    print("Epoch:", '%04d' % (epoch+1),
          "cost=", "{:.9f}".format(loss_value))

print("Optimization Finished!")
```

```
Epoch: 0001 cost= 0.086695343
Epoch: 0002 cost= 0.086695544
Epoch: 0003 cost= 0.086692639
Epoch: 0004 cost= 0.086690240
Epoch: 0005 cost= 0.086685091
Epoch: 0006 cost= 0.086675093
Epoch: 0007 cost= 0.086650193
Epoch: 0008 cost= 0.086561024
Epoch: 0009 cost= 0.084797345
Epoch: 0010 cost= 0.074700832
Epoch: 0011 cost= 0.019813314
Epoch: 0012 cost= 0.017553935
Epoch: 0013 cost= 0.015318388
Epoch: 0014 cost= 0.016993877
Epoch: 0015 cost= 0.015222666
Epoch: 0016 cost= 0.014809877
Epoch: 0017 cost= 0.015596349
Epoch: 0018 cost= 0.014048850
Epoch: 0019 cost= 0.013970785
Epoch: 0020 cost= 0.013903070
Epoch: 0021 cost= 0.014142765
Epoch: 0022 cost= 0.014070742
Epoch: 0023 cost= 0.013419215
Epoch: 0024 cost= 0.013191053
Epoch: 0025 cost= 0.013080104
Epoch: 0026 cost= 0.012979521
Epoch: 0027 cost= 0.012839882
Epoch: 0028 cost= 0.012634150
Epoch: 0029 cost= 0.012843819
Epoch: 0030 cost= 0.012478763
Epoch: 0031 cost= 0.012552181
Epoch: 0032 cost= 0.012422295
Epoch: 0033 cost= 0.012395135
Epoch: 0034 cost= 0.012341714
Epoch: 0035 cost= 0.012313658
Epoch: 0036 cost= 0.012271455
Epoch: 0037 cost= 0.012243968
Epoch: 0038 cost= 0.012227491
Epoch: 0039 cost= 0.012206021
Epoch: 0040 cost= 0.012163056
Epoch: 0041 cost= 0.012133112
Epoch: 0042 cost= 0.012110059
Epoch: 0043 cost= 0.012087669
Epoch: 0044 cost= 0.012063604
Epoch: 0045 cost= 0.012040107
Epoch: 0046 cost= 0.012011886
Epoch: 0047 cost= 0.011994647
Epoch: 0048 cost= 0.011960381
Epoch: 0049 cost= 0.011950091
Epoch: 0050 cost= 0.011921189
Optimization Finished!
```

```
model.summary()
```

```
Model: "auto_encoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         multiple                  0 (unused)

 dense_6 (Dense)             multiple                  192256

 dense_7 (Dense)             multiple                  32896

 dense_8 (Dense)             multiple                  4128

 dense_9 (Dense)             multiple                  4224

 dense_10 (Dense)            multiple                  33024

 dense_11 (Dense)            multiple                  192750

=================================================================
Total params: 459,278
Trainable params: 459,278
Non-trainable params: 0
_________________________________________________________________
```

```
from tensorflow.python.ops.numpy_ops import np_config
np_config.enable_numpy_behavior()
```

```
Data[0].reshape(1,-1)
```

```
        0.6415712 , 0.14353313, 0.80705625, 0.65302783, 0.13249211,
        0.79272324, 0.66775775, 0.14353313, 0.78280044, 0.6759411 ,
        0.1466877 , 0.7673649 , 0.68903434, 0.13722397, 0.75413454,
        0.705401  , 0.14037855, 0.7442117 , 0.7184943 , 0.12618296,
        0.7342889 , 0.7283142 , 0.13249211, 0.72657114, 0.7397709 ,
        0.12460568, 0.7166483 , 0.7528642 , 0.12302839, 0.7056229 ,
        0.7675941 , 0.1214511 , 0.6990077 , 0.7839607 , 0.11514196,
        0.6879824 , 0.79541737, 0.11829653, 0.6780595 , 0.80360067,
        0.11198738, 0.6736494 , 0.81178397, 0.10883281, 0.6582139 ,
        0.82978725, 0.10725552, 0.64829105, 0.8396072 , 0.10725552,
        0.6405733 , 0.84942716, 0.11041009, 0.631753  , 0.85433716,
        0.10567823, 0.6262404 , 0.85761046, 0.10252366, 0.6185226 ,
        0.86743045, 0.10725552, 0.6130099 , 0.87397707, 0.10725552,
        0.6085998 , 0.87561375, 0.40536278, 0.19514884, 0.9459902 ,
        0.40851736, 0.18853363, 0.9410802 , 0.42113563, 0.17750826,
        0.9410802 , 0.41955835, 0.1719956 , 0.9394435 , 0.43217665,
        0.16648291, 0.93780684, 0.43533123, 0.16427784, 0.92962354,
        0.4400631 , 0.15876517, 0.9312602 , 0.4400631 , 0.15435502,
        0.9328969 , 0.46056783, 0.14994487, 0.9198036 , 0.46056783,
        0.14663726, 0.9198036 , 0.46687698, 0.14443219, 0.9148936 ,
        0.47003156, 0.14112459, 0.91653025, 0.48264983, 0.13891952,
        0.90998363, 0.48264983, 0.13671444, 0.90507364, 0.49684542,
        0.13340683, 0.90016365, 0.48895898, 0.14112459, 0.88379705,
        0.50473183, 0.13891952, 0.8477905 , 0.5094637 , 0.13891952,
        0.87070376, 0.51419556, 0.14112459, 0.86579376, 0.52681386,
        0.13671444, 0.84615386, 0.52996844, 0.13671444, 0.84288055,
        0.54258674, 0.13891952, 0.8314239 , 0.5504732 , 0.1356119 ,
        0.8363339 , 0.5646688 , 0.13450937, 0.81996727, 0.5646688 ,
        0.1323043 , 0.81178397, 0.57886434, 0.13009922, 0.8052373 ,
        0.5899054 , 0.12679163, 0.79541737, 0.5993691 , 0.12568909,
        0.7774141 , 0.61356467, 0.12458655, 0.7725041 , 0.62460566,
        0.12127894, 0.7561375 , 0.6388013 , 0.11797133, 0.7414075 ,
        0.64984226, 0.12127894, 0.7332242 , 0.66561514, 0.11907387,
        0.7266776 , 0.6766561 , 0.11576626, 0.7070376 , 0.68611985,
        0.11135612, 0.695581  , 0.70189273, 0.10584344, 0.690671  ,
        0.71924293, 0.10253583, 0.68085104, 0.7287066 , 0.10033076,
        0.6710311 , 0.7492114 , 0.09702315, 0.65957445, 0.89432174,
        0.32414553, 0.28968903, 0.8895899 , 0.33406836, 0.27168575,
        0.89905363, 0.34619626, 0.26022914, 0.90378547, 0.3572216 ,
        0.24549918, 0.9022082 , 0.37155458, 0.23076923, 0.90378547,
        0.3792723 , 0.2111293 , 0.90851736, 0.38919514, 0.20458265,
        0.9100946 , 0.40132305, 0.18821605, 0.9132492 , 0.41345093,
        0.18657938, 0.9164038 , 0.41786107, 0.18003273, 0.9132492 ,
        0.4189636 , 0.17348608, 0.9132492 , 0.42888644, 0.16693944,
        0.90694004, 0.43439913, 0.16693944, 0.90851736, 0.4432194 ,
        0.16693944, 0.90694004, 0.44432193, 0.15384616, 0.9022082 ,
        0.44432193, 0.15875614, 0.9022082 , 0.4553473 , 0.1603928 ,
        0.89589906, 0.45975745, 0.1603928 , 0.89905363, 0.46857774,
        0.15384616, 0.89116716, 0.47298786, 0.15220949, 0.88801265,
        0.477398  , 0.15057284, 0.8848581 , 0.48511577, 0.15057284,
        0.88328075, 0.491731  , 0.14729951, 0.88643533, 0.49724367,
        0.14566284, 0.8769716 , 0.50496143, 0.14729951, 0.8690852 ,
        0.51267916, 0.14402619, 0.85804415, 0.5181918 , 0.14729951,
        0.851735  , 0.5270121 , 0.14566284, 0.840694  , 0.5380375 ,
        0.14893617, 0.8312303 , 0.5457552 , 0.14893617, 0.82176656,
        0.55567807, 0.16202946, 0.8154574 , 0.56449836, 0.15220949,
        0.807571  , 0.57662624, 0.14729951, 0.79810727, 0.58654904]],
      dtype=float32)>
```

```
encode_decode = model(Data[0].reshape(1,-1))
encode_decode
```

```
0.2861219 , 0.73065937, 0.45512855, 0.27205512, 0.7403887 ,
0.48792154, 0.25081506, 0.7311151 , 0.47662947, 0.23711306,
0.75337785, 0.5200805 , 0.23451379, 0.7220156 , 0.53734154,
0.22636151, 0.7276686 , 0.53090364, 0.2239581 , 0.6927159 ,
0.55979025, 0.20191981, 0.6861197 , 0.56778276, 0.21191245,
0.6566323 , 0.5932726 , 0.21046641, 0.6765121 , 0.6054281 ,
0.19428927, 0.63912404, 0.6028828 , 0.21082398, 0.64043164,
0.6146731 , 0.17933701, 0.62603796, 0.6332479 , 0.21009406,
0.620464  , 0.6492695 , 0.21482512, 0.61530066, 0.64876705,
0.19546758, 0.6092506 , 0.65449107, 0.19525738, 0.58299863,
0.665898  , 0.21186528, 0.5559174 , 0.68209505, 0.2352606 ,
0.55329776, 0.6708972 , 0.21985498, 0.5204699 , 0.6900872 ,
0.21064492, 0.5288511 , 0.70070755, 0.24272236, 0.52400464,
0.7093173 , 0.24549803, 0.51585186, 0.73810565, 0.25082138,
0.48627946, 0.69763327, 0.26428583, 0.46137974, 0.7164358 ,
0.2940967 , 0.43372536, 0.7054741 , 0.30762446, 0.4438412 ,
0.7191012 , 0.29730693, 0.44527292, 0.73703134, 0.31887037,
0.433166  , 0.7141139 , 0.3280835 , 0.41858926, 0.70521516,
0.33843148, 0.4100553 , 0.71511406, 0.3470367 , 0.39072222,
0.71476597, 0.36347604, 0.3987956 , 0.7311233 , 0.35704416,
0.37273073, 0.73342   , 0.367009  , 0.37784702, 0.7411709 ,
0.36941272, 0.33783808, 0.74038243, 0.3816642 , 0.34444842,
0.7461415 , 0.3991531 , 0.3258505 , 0.7352934 , 0.39295352,
0.3208341 , 0.7314604 , 0.4167474 , 0.32409167, 0.744913  ,
0.42011547, 0.30849478, 0.7170514 , 0.42173046, 0.2789697 ,
0.7400675 , 0.4134211 , 0.2724046 , 0.74713606, 0.43731755,
0.2447967 , 0.7390838 , 0.47102925, 0.2519229 , 0.7294352 ,
0.50411475, 0.22024265, 0.7270978 , 0.5097175 , 0.23531856,
0.7187604 , 0.5262727 , 0.23504205, 0.6899461 , 0.5496372 ,
0.21699943, 0.7057334 , 0.5647577 , 0.21199775, 0.69291866,
0.55895704, 0.20288281, 0.67912924, 0.57361245, 0.1943636 ,
0.6884582 , 0.57729447, 0.22152184, 0.6763876 , 0.5975316 ,
```

```
E = model(Data)
```

```
E.numpy().reshape(-1,3).shape
```

```
(3170000, 3)
```

```
E = pd.DataFrame(E.numpy().reshape(-1,3))
```

```
D = pd.DataFrame(data.reshape(-1,3))
```

```
D.shape, E.shape
```

```
((3170000, 3), (3170000, 3))
```

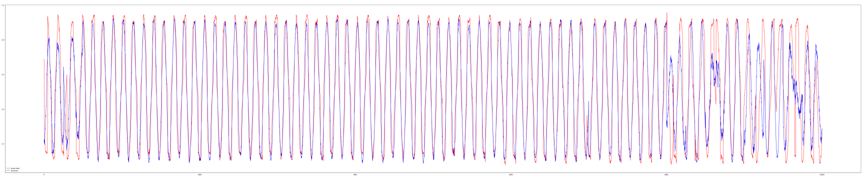
```
model_json = model.to_json()
with open("/content/drive/MyDrive/data/model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("/content/drive/MyDrive/data/model.h5")
print("Saved model")
```

```
Saved model
```

```
from tensorflow.keras.models import model_from_json

json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")
```

```
fig = plt.figure(figsize=(400,80))
for i,j in zip(range(317),range(1,318)):
  plt.subplot(80,4,j)
  plt.plot(range(10000),D[0][i*10000:(i+1)*10000],color='red') # actual data
  plt.plot(range(10000),E[0][i*10000:(i+1)*10000],color='blue') # predicted

plt.show()
```

```
fig = plt.figure(figsize=(100,20))
plt.plot(range(10000),D[0][0*10000:1*10000],color='red') # actual data
plt.plot(range(10000),E[0][0*10000:1*10000],color='blue') # predicted
plt.legend(['actual data','predicted'],loc='lower left')
plt.show()
```



✓ 12s completed at 5:19 PM