Artificial neural network

Import Libraries

In [1]:

```
import pandas as pd
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import seaborn as sns
import pickle
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from keras.models import model_from_json
import pydot_ng as pydot
from keras.utils import plot_model
import time
```

```
Using TensorFlow backend.
```

Load data

In [2]:

```
df = pd.read_csv('sub1.csv', low_memory=False)
```

Below are the first 5 rows of the data.

In [3]:

```
df.head()
```

Out[3]:

	0	1	2	3	4
0	-65.746971	-66.827461	-65.604935	-63.861546	-66.703033
1	-66.542122	-67.348610	-66.031960	-64.742157	-66.823357
2	-66.703415	-67.738510	-68.254852	-65.018089	-66.908104
3	-67.461182	-68.072807	-68.260017	-66.640083	-66.984612
4	-68.103065	-68.076118	-68.277176	-66.681175	-67.161224

In [4]:

```
df.columns
```

Out[4]:

```
Index(['0', '1', '2', '3', '4'], dtype='object')
```

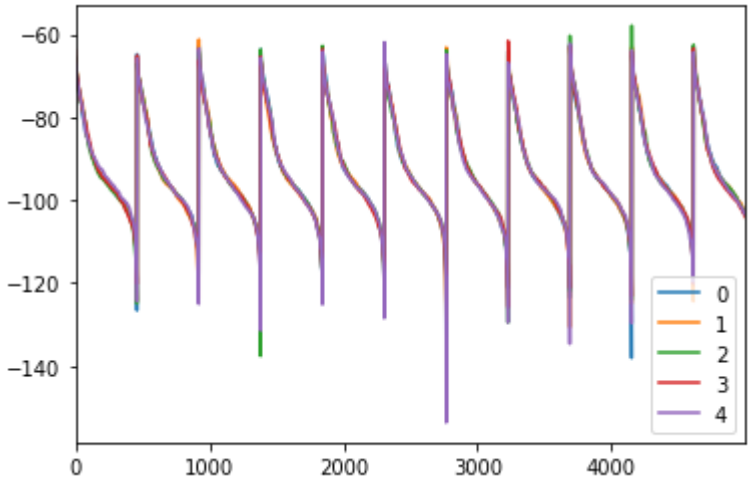
Data visualisation of the first 5000 rows

In [5]:

```
df[:5000].plot()
```

Out[5]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x4088ecf908>
```



from the graph above, the values are the almost the same across all columns.so we can use one column to represent all thereby avoiding complexity

In [6]:

```
df['selected'] = df.iloc[:,2]
```

Here, we picked one column to represent the data since they are pretty much alike

Normalisation

- this helps the model work better as they dont understand the context of the data, ages of people and heights of people are different context which the model cant understand. normalisation gives the values a particular to help the model understand the limits and context of the data.
- we also performed some splitting activities to divide the data so we can test the model with a data that it has not seen.

In [7]:

```
dataset     = df['selected'].values #numpy.ndarray
dataset     = dataset.astype('float32')
dataset     = np.reshape(dataset, (-1, 1))
scaler      = MinMaxScaler(feature_range=(0, 1))
dataset     = scaler.fit_transform(dataset)
train_size  = int(len(dataset) * 0.80)
test_size   = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
```

Feature extraction using timesteps

We are using the previous data values to determine the next one.

so the value at (t-1)s will be used to determine value at (t)s

we are employing 12 timesteps, so the previous 12 values will be used to determine the 13th values.

In [8]:

```
def create_dataset(dataset, look_back=1):
    X, Y = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        X.append(a)
        Y.append(dataset[i + look_back, 0])
    return np.array(X), np.array(Y)
```

In [9]:

```
n_steps       = 12
x,y           = create_dataset(train, n_steps)
x_test,y_test = create_dataset(test, n_steps)
```

Building ANN model

In [10]:

```
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps))
model.add(Dense(80))
model.add(Dense(50))
model.add(Dense(50))
model.add(Dropout(0.2))
model.add(Dense(50))
model.add(Dense(40))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

In [11]:

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 100)               1300
_________________________________________________________________
dense_2 (Dense)              (None, 80)                8080
_________________________________________________________________
dense_3 (Dense)              (None, 50)                4050
_________________________________________________________________
dense_4 (Dense)              (None, 50)                2550
_________________________________________________________________
dropout_1 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_5 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_6 (Dense)              (None, 40)                2040
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 41
=================================================================
Total params: 20,611
Trainable params: 20,611
Non-trainable params: 0
_________________________________________________________________
```

Training the model

In [12]:

```
start=time.time()
history = model.fit(x, y,
                    batch_size=100,
                    epochs=20,
                    verbose=2,
                    validation_data=(x_test, y_test))
end=time.time()
training_time=end-start
print("Training time:",training_time)
```

```
Train on 1514346 samples, validate on 378577 samples
Epoch 1/20
 - 43s - loss: 9.0720e-04 - val_loss: 3.5686e-04
Epoch 2/20
 - 41s - loss: 6.7577e-04 - val_loss: 3.4645e-04
Epoch 3/20
 - 42s - loss: 6.5550e-04 - val_loss: 3.6933e-04
Epoch 4/20
 - 42s - loss: 6.4598e-04 - val_loss: 3.7358e-04
Epoch 5/20
 - 41s - loss: 6.4016e-04 - val_loss: 3.6275e-04
Epoch 6/20
 - 41s - loss: 6.3504e-04 - val_loss: 3.5099e-04
Epoch 7/20
 - 41s - loss: 6.3023e-04 - val_loss: 3.5474e-04
Epoch 8/20
 - 40s - loss: 6.2841e-04 - val_loss: 3.4274e-04
Epoch 9/20
 - 40s - loss: 6.2759e-04 - val_loss: 3.3721e-04
Epoch 10/20
 - 40s - loss: 6.2585e-04 - val_loss: 3.6224e-04
Epoch 11/20
 - 40s - loss: 6.2506e-04 - val_loss: 3.4465e-04
Epoch 12/20
 - 41s - loss: 6.2413e-04 - val_loss: 3.6491e-04
Epoch 13/20
 - 41s - loss: 6.2191e-04 - val_loss: 3.6103e-04
Epoch 14/20
 - 40s - loss: 6.2060e-04 - val_loss: 3.3398e-04
Epoch 15/20
 - 40s - loss: 6.2046e-04 - val_loss: 3.5104e-04
Epoch 16/20
 - 41s - loss: 6.1850e-04 - val_loss: 3.3494e-04
Epoch 17/20
 - 41s - loss: 6.1673e-04 - val_loss: 3.6814e-04
Epoch 18/20
 - 41s - loss: 6.1710e-04 - val_loss: 3.4809e-04
Epoch 19/20
 - 42s - loss: 6.1570e-04 - val_loss: 3.4556e-04
Epoch 20/20
 - 41s - loss: 6.1613e-04 - val_loss: 3.6513e-04
Training time: 823.0135462284088
```

Predicting the values

In [13]:

```
start=time.time()
train_predict = model.predict(x)
test_predict = model.predict(x_test)
# invert predictions
train_predict = scaler.inverse_transform(train_predict)
Y_train = scaler.inverse_transform([y])
test_predict = scaler.inverse_transform(test_predict)
Y_test = scaler.inverse_transform([y_test])
end=time.time()
prediction_time=end-start
print("Prediction time: ",prediction_time)
```

```
Prediction time:  27.539127826690674
```

Measuring accuracy of the model

there is no way to use percentage accuracies to determine accuracies, instead we use MSE mean squared error, MASE mean absolute squared error. this errors simply show how far the the predicted values are from the actual values to show their competency when faced with real-world data.

In [14]:

```
print('Train Mean Absolute Error:', mean_absolute_error(Y_train[0], train_predict[:,0]))
print('Train Root Mean Squared Error:',np.sqrt(mean_squared_error(Y_train[0], train_predict
print('Test Mean Absolute Error:', mean_absolute_error(Y_test[0], test_predict[:,0]))
print('Test Root Mean Squared Error:',np.sqrt(mean_squared_error(Y_test[0], test_predict[:,
```

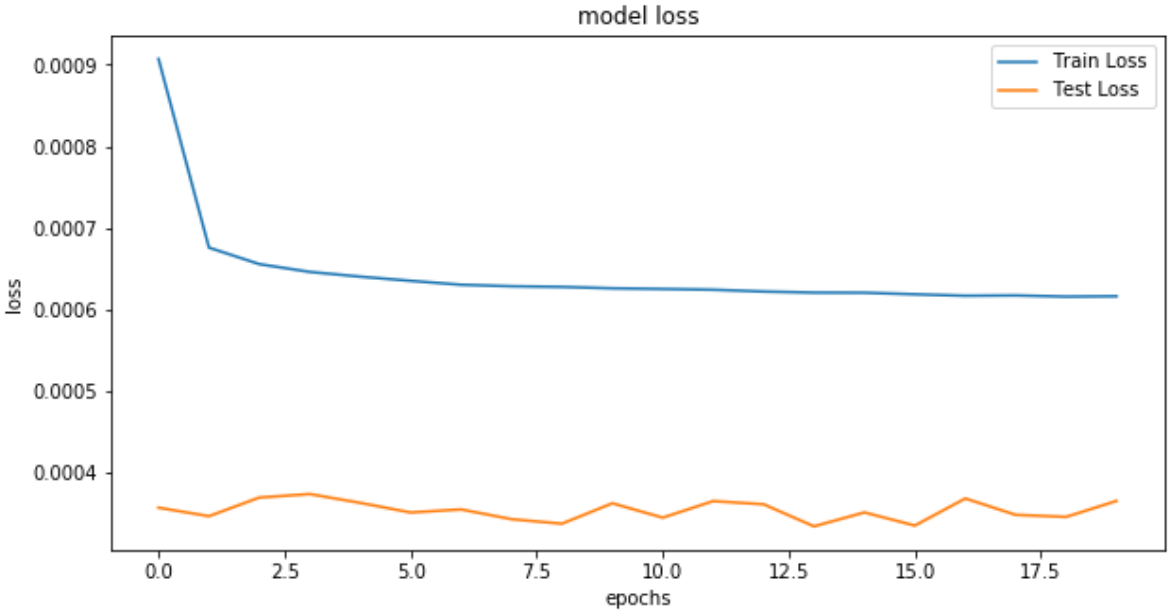
```
Train Mean Absolute Error: 0.9607954577649176
Train Root Mean Squared Error: 2.8690985590189575
Test Mean Absolute Error: 0.7121428560653249
Test Root Mean Squared Error: 2.2580241878082075
```

Model loss

the graph shows the errors when training and validating. it is another metric that shows the reliability of the system. the validation error should not be higher than training loss because, the validation loss is a loss when exposed to a new data and training loss is the loss when exposed to the data that was used to train the model. When this happens, it is similar to the term we call 'CRAMMING'.

In [15]:

```
plt.figure(figsize=(10,5))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Test Loss')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epochs')
plt.legend(loc='upper right')
plt.show()
```

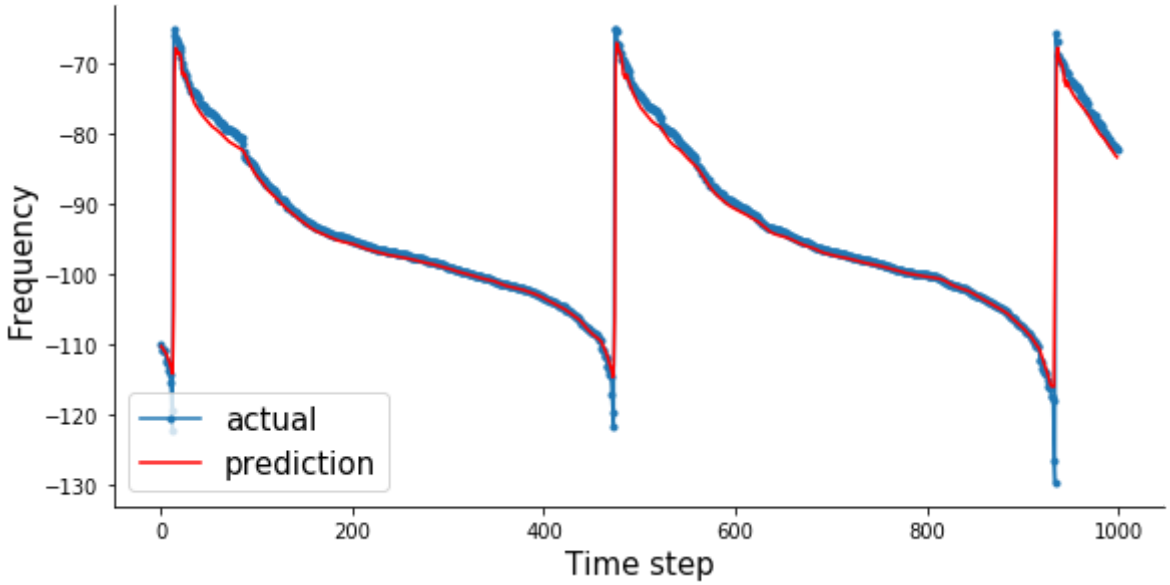


Graph of the Actual data vs the predicted

This shows how different the predicted data is from the actual data

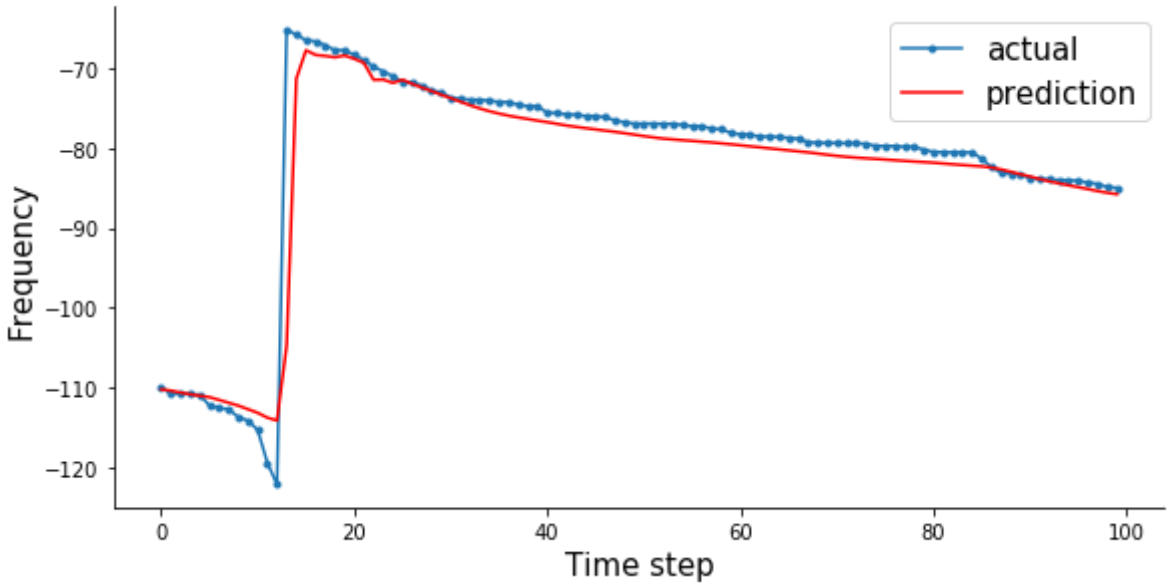
In [16]:

```
aa=[x for x in range(1000)]
plt.figure(figsize=(8,4))
plt.plot(aa, Y_test[0][:1000], marker='.', label="actual")
plt.plot(aa, test_predict[:,0][:1000], 'r', label="prediction")
# plt.tick_params(left=False, labelleft=True) #remove ticks
plt.tight_layout()
sns.despine(top=True)
plt.subplots_adjust(left=0.07)
plt.ylabel('Frequency', size=15)
plt.xlabel('Time step', size=15)
plt.legend(fontsize=15)
plt.show()
```



In [17]:

```
aa=[x for x in range(100)]
plt.figure(figsize=(8,4))
plt.plot(aa, Y_test[0][:100], marker='.', label="actual")
plt.plot(aa, test_predict[:,0][:100], 'r', label="prediction")
# plt.tick_params(left=False, labelleft=True) #remove ticks
plt.tight_layout()
sns.despine(top=True)
plt.subplots_adjust(left=0.07)
plt.ylabel('Frequency', size=15)
plt.xlabel('Time step', size=15)
plt.legend(fontsize=15)
plt.show()
```



Saving predicted data

In [18]:

```
sed=pd.DataFrame(test_predict)
sed.to_csv('imputed_improved_DNN_predict.csv', index=False)
```

Saving the model

In [19]:

```
filename = 'imputed_improved_DNN_model.sav'
pickle.dump(model, open(filename, 'wb'))
```

In [20]:

```
model_json = model.to_json()
with open("imputed_improved_DNN_model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")
```

```
Saved model to disk
```

Saving the normalisation model

the ANN model was built on normalised data for better result. When attempting to predict new values, the data must be normalised before being sent to the model

In [21]:

```
filename = 'imputed_improved_DNN_scaler.sav'
pickle.dump(scaler, open(filename, 'wb'))
```

Loading the saved model

In [22]:

```
json_file = open('imputed_improved_DNN_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")
```

```
Loaded model from disk
```

In [23]:

```
look_back = 12
X_train, Y_train = create_dataset(train, look_back)
X_test, Y_test = create_dataset(test, look_back)
#X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))
#X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))
train_predict = loaded_model.predict(X_train)
test_predict = loaded_model.predict(X_test)
# invert predictions
train_predict = scaler.inverse_transform(train_predict)
Y_train = scaler.inverse_transform([Y_train])
test_predict = scaler.inverse_transform(test_predict)
Y_test = scaler.inverse_transform([Y_test])
print('Train Mean Absolute Error:', mean_absolute_error(Y_train[0], train_predict[:,0]))
print('Train Root Mean Squared Error:',np.sqrt(mean_squared_error(Y_train[0], train_predict
print('Test Mean Absolute Error:', mean_absolute_error(Y_test[0], test_predict[:,0]))
print('Test Root Mean Squared Error:',np.sqrt(mean_squared_error(Y_test[0], test_predict[:,
```

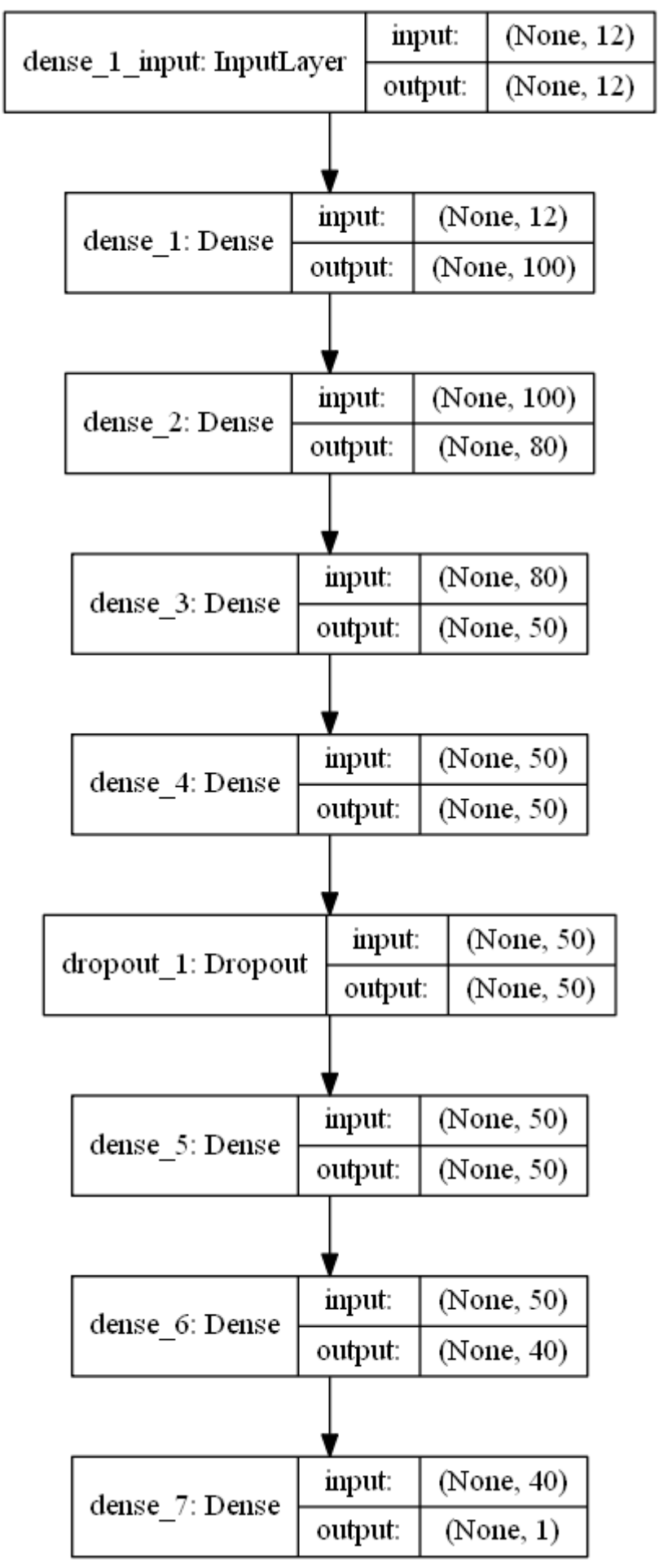
```
Train Mean Absolute Error: 0.9607954577649176
Train Root Mean Squared Error: 2.8690985590189575
Test Mean Absolute Error: 0.7121428560653249
Test Root Mean Squared Error: 2.2580241878082075
```

Visualisation of the Neural Network

In [24]:

```
plot_model(model, to_file='imputed_improved_DNN.png', show_shapes=True)
```

Out[24]:



Model Parameters

- Epoch = 20

.

- Batch size = 100
- Hidden layers = 6
- Dropout layers = 1
- Dropout = 0.2
- Neurons = 100, 80, 50, 50, 50, 40 in hidden layers respectively, 1 in output layer

Observation	values
Train Mean Absolute Error	0.9607
Train Root Mean Squared Error	2.8690
Test Mean Absolute Error	0.7121
Test Root Mean Squared Error	2.2580
Training time	823.0135
Prediction time	27.5391

Observation	DNN	Improved DNN
Train Mean Absolute Error	1.0161	0.9607
Train Root Mean Squared Error	2.8540	2.8690
Test Mean Absolute Error	0.7674	0.7121
Test Root Mean Squared Error	2.2315	2.2580
Training time	385.8252	823.0135
Prediction time	16.4987	27.5391

Models	Train Mean Absolute Error	Train Root Mean Squared Error	Test Mean Absolute Error	Test Root Mean Squared Error	Training time	Prediction time
DNN	1.0161	2.8540	0.7674	2.2315	385.8252	16.4987
Improved DNN	0.9607	2.8690	0.7121	2.2580	823.0135	27.5391
LSTM	0.9682	2.9601	0.7071	2.3209	1445.3598	40.5545
Improved LSTM	0.6871	2.8156	0.4220	2.1649	2062.3754	56.5365