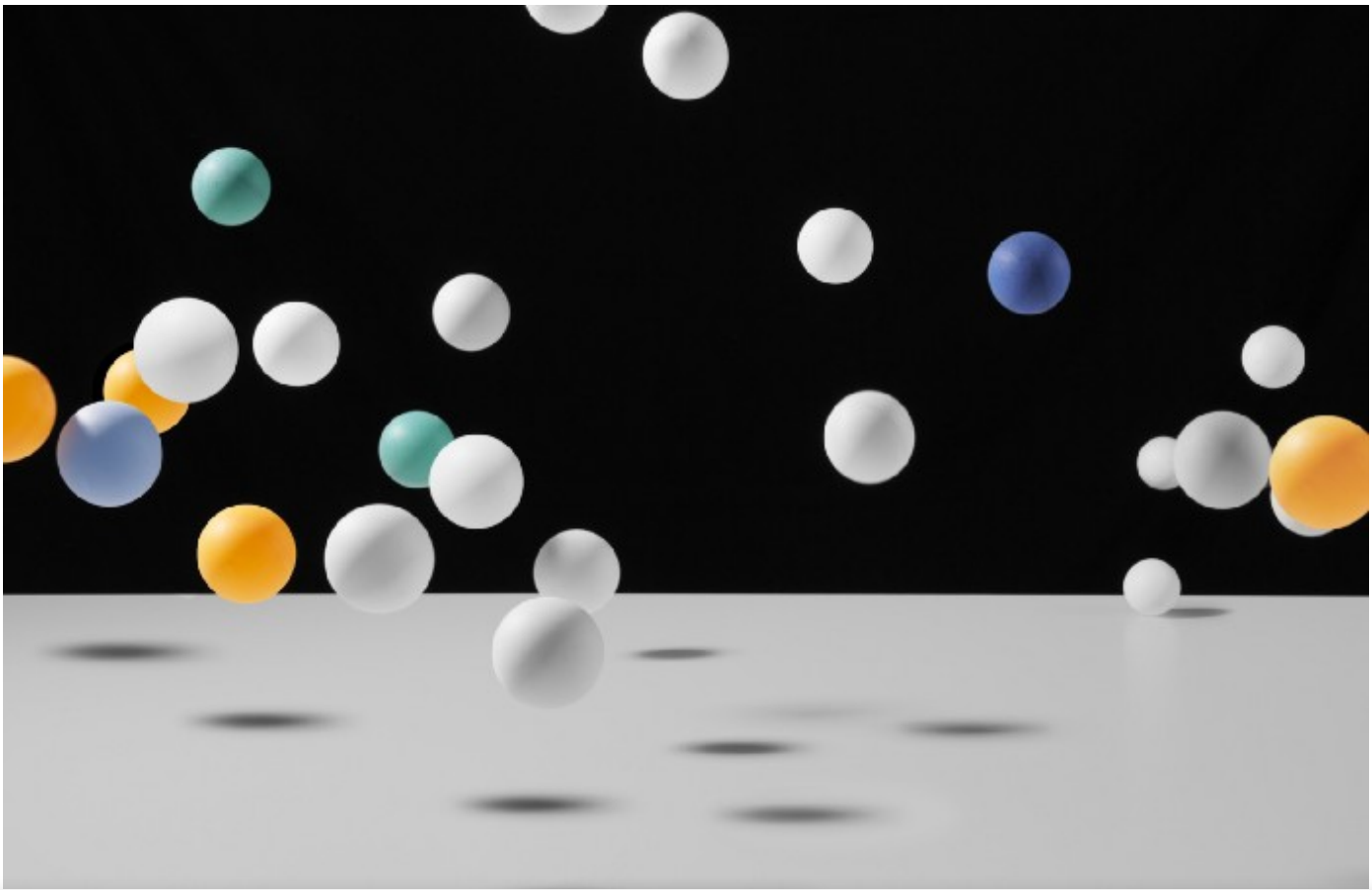
Published in Towards Data Science

Brent Larzalere Follow

Sep 25, 2019 · 8 min read ★ · Listen

LSTM Autoencoder for Anomaly Detection

Create an AI deep learning anomaly detection model using Python, Keras and TensorFlow



The goal of this post is to walk you through the steps to create and train an AI deep learning neural network for anomaly detection using Python, Keras and TensorFlow. I will not delve too much in to the underlying theory and assume the reader has some basic knowledge of the underlying technologies. However, I will provide links to more detailed information as we go and you can find the source code for this study in my GitHub repo.

Analysis Dataset

We will use vibration sensor readings from the NASA Acoustics and Vibration Database as our dataset for this study. In the NASA study, sensor readings were taken on four bearings that were run to failure under constant load over multiple days. Our dataset consists of individual files that are 1-second vibration signal snapshots recorded at 10 minute intervals. Each file contains 20,480 sensor data points per bearing that were obtained by reading the bearing sensors at a sampling rate of 20 kHz.

You can download the sensor data here. Due to GitHub size limitations, the bearing sensor data is split between two zip files (Bearing_Sensor_Data_pt1 and 2). You will need to unzip them and combine them into a single data directory.

Anomaly Detection

Anomaly detection is the task of determining when something has gone astray from the "norm". Anomaly detection using neural networks is modeled in an unsupervised / self-supervised manner; as opposed to supervised learning, where there is a one-to-one correspondence between input feature samples and their corresponding output labels. The presumption is that normal behavior, and hence the quantity of available "normal" data, is the norm and that anomalies are the exception to the norm to the point where the modeling of "normalcy" is possible.

We will use an autoencoder deep learning neural network model to identify vibrational anomalies from the sensor readings. The goal is to predict future bearing failures before they happen.

LSTM Networks


networks are a sub-type of the more general recurrent neural networks (RNN). A key attribute of recurrent neural networks is their ability to persist information, or cell state, for use later in the network. This makes them particularly well suited for analysis of temporal data that evolves over time. LSTM networks are used in tasks such as speech recognition, text translation and here, in the analysis of sequential sensor readings for anomaly detection.

There are numerous excellent articles by individuals far better qualified than I to discuss the fine details of LSTM networks. So if you're curious, here is a link to an excellent article on LSTM networks. There is also the defacto place for all things LSTM — Andrej Karpathy's blog. Enough with the theory, let's get on with the code…

Load , Pre-Process & Review Data

I will be using an Anaconda distribution Python 3 Jupyter notebook for creating and training our neural network model. We will use TensorFlow as our backend and Keras as our core model development library. The first task is to load our Python libraries. We then set our random seed in order to create reproducible results.

```
# import libraries
import os
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.externals import joblib
import seaborn as sns
sns.set(color_codes=True)
import matplotlib.pyplot as plt
%matplotlib inline

from numpy.random import seed
from tensorflow import set_random_seed
import tensorflow as tf
tf.logging.set_verbosity(tf.logging.ERROR)


from keras.layers import Input, Dropout, Dense, LSTM, TimeDistributed, RepeatVector
from keras.models import Model
from keras import regularizers
```

```
# set random seed
seed(10)
set_random_seed(10)
```

vibration recordings over the 20,480 datapoints. We then merge everything together into a single Pandas dataframe.

```
# load, average and merge sensor samples
data_dir = 'data/bearing_data'
merged_data = pd.DataFrame()

for filename in os.listdir(data_dir):
    dataset = pd.read_csv(os.path.join(data_dir, filename), sep='\t')
    dataset_mean_abs = np.array(dataset.abs().mean())
    dataset_mean_abs = pd.DataFrame(dataset_mean_abs.reshape(1,4))
    dataset_mean_abs.index = [filename]
    merged_data = merged_data.append(dataset_mean_abs)

merged_data.columns = ['Bearing 1', 'Bearing 2', 'Bearing 3', 'Bearing 4']
```

```
# transform data file index to datetime and sort in chronological order
merged_data.index = pd.to_datetime(merged_data.index, format='%Y.%m.%d.%H.%M.%S')
merged_data = merged_data.sort_index()
merged_data.to_csv('Averaged_BearingTest_Dataset.csv')
print("Dataset shape:", merged_data.shape)
merged_data.head()
```

```
Dataset shape: (982, 4)
```

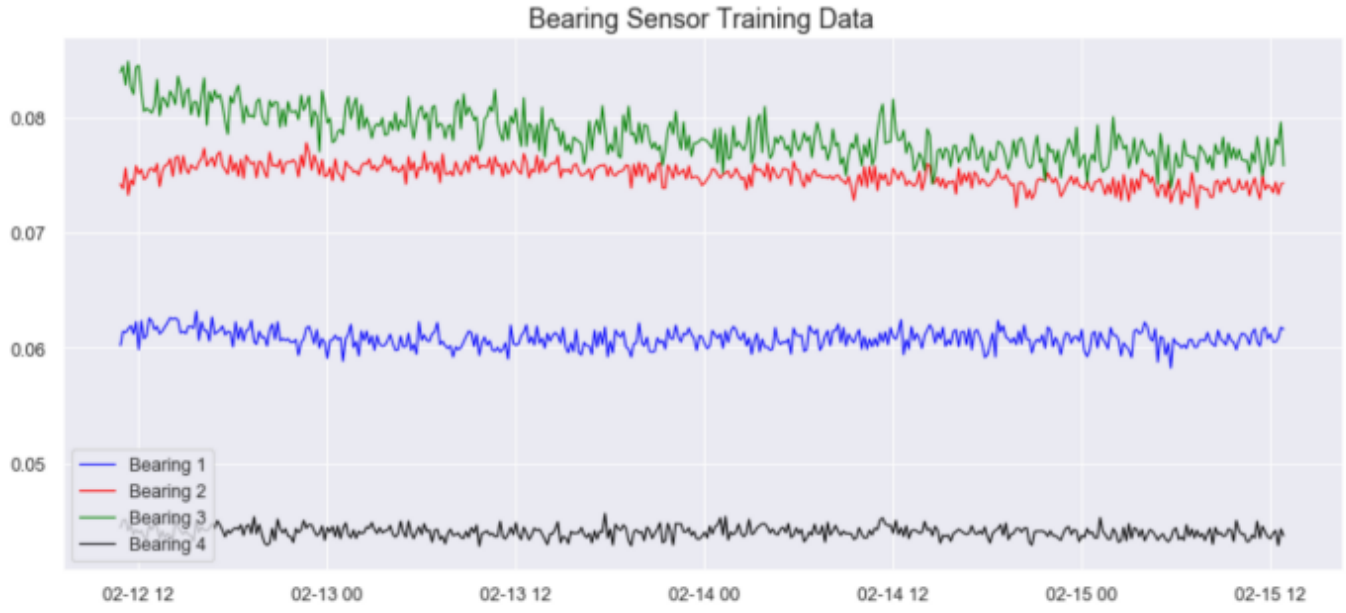
	Bearing 1	Bearing 2	Bearing 3	Bearing 4
2004-02-12 10:52:39	0.060236	0.074227	0.083926	0.044443
2004-02-12 11:02:39	0.061455	0.073844	0.084457	0.045081
2004-02-12 11:12:39	0.061361	0.075609	0.082837	0.045118
2004-02-12 11:22:39	0.061665	0.073279	0.084879	0.044172
2004-02-12 11:32:39	0.061944	0.074593	0.082626	0.044659

Next, we define the datasets for training and testing our neural network. To do this, we perform a simple split where we train on the first part of the dataset, which represents normal operating conditions. We then test on the remaining part of the dataset that contains the sensor readings leading up to the bearing failure.

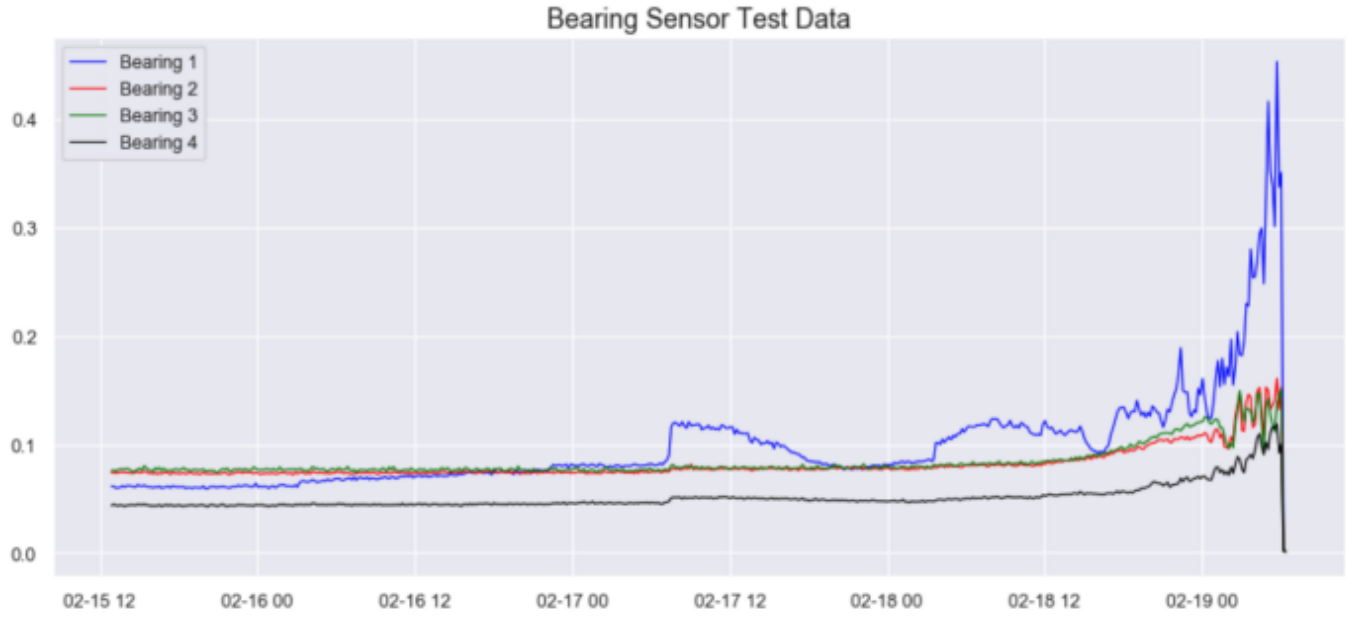
```
train = merged_data['2004-02-12 10:52:39': '2004-02-15 12:52:39']
test = merged_data['2004-02-15 12:52:39':]
print("Training dataset shape:", train.shape)
print("Test dataset shape:", test.shape)
```

```
Training dataset shape: (445, 4)
Test dataset shape: (538, 4)
```

```
fig, ax = plt.subplots(figsize=(14, 6), dpi=80)
ax.plot(train['Bearing 1'], label='Bearing 1', color='blue', animated = True, linewidth=1)
ax.plot(train['Bearing 2'], label='Bearing 2', color='red', animated = True, linewidth=1)
ax.plot(train['Bearing 3'], label='Bearing 3', color='green', animated = True, linewidth=1)
ax.plot(train['Bearing 4'], label='Bearing 4', color='black', animated = True, linewidth=1)
plt.legend(loc='lower left')
ax.set_title('Bearing Sensor Training Data', fontsize=16)
plt.show()
```



Next, we take a look at the test dataset sensor readings over time.

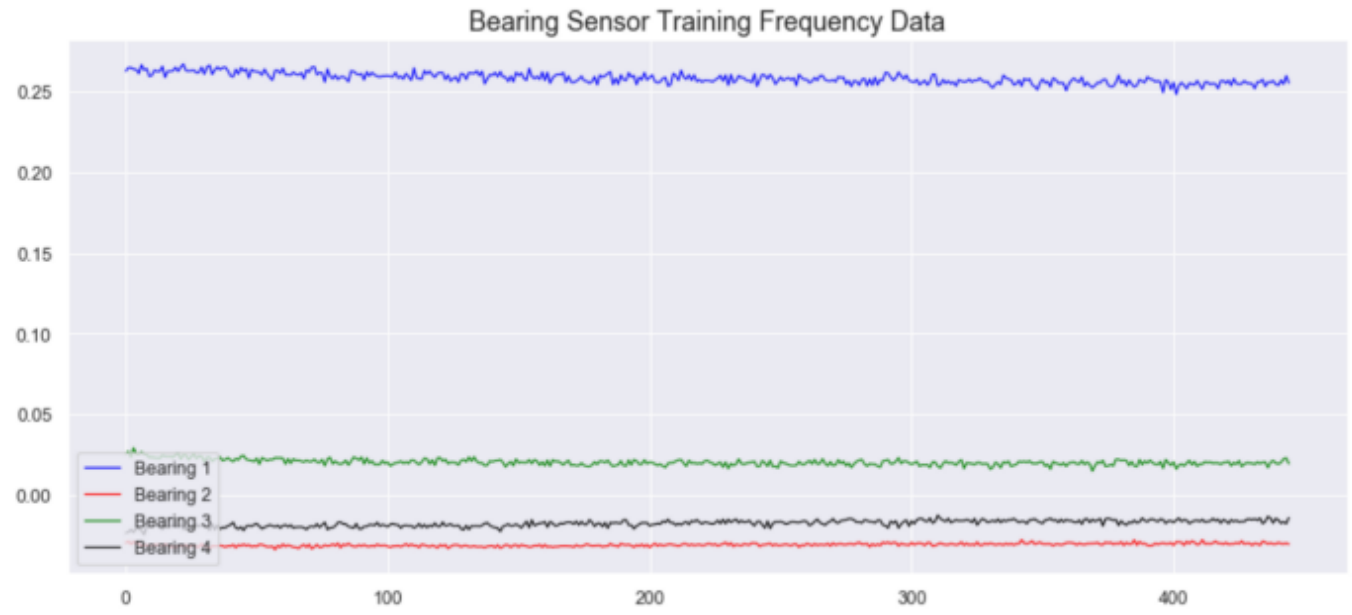


To gain a slightly different perspective of the data, we will transform the signal from the time domain to the frequency domain using a Fourier transform.

```
# transforming data from the time domain to the frequency domain using fast Fourier transform
train_fft = np.fft.fft(train)
test_fft = np.fft.fft(test)
```

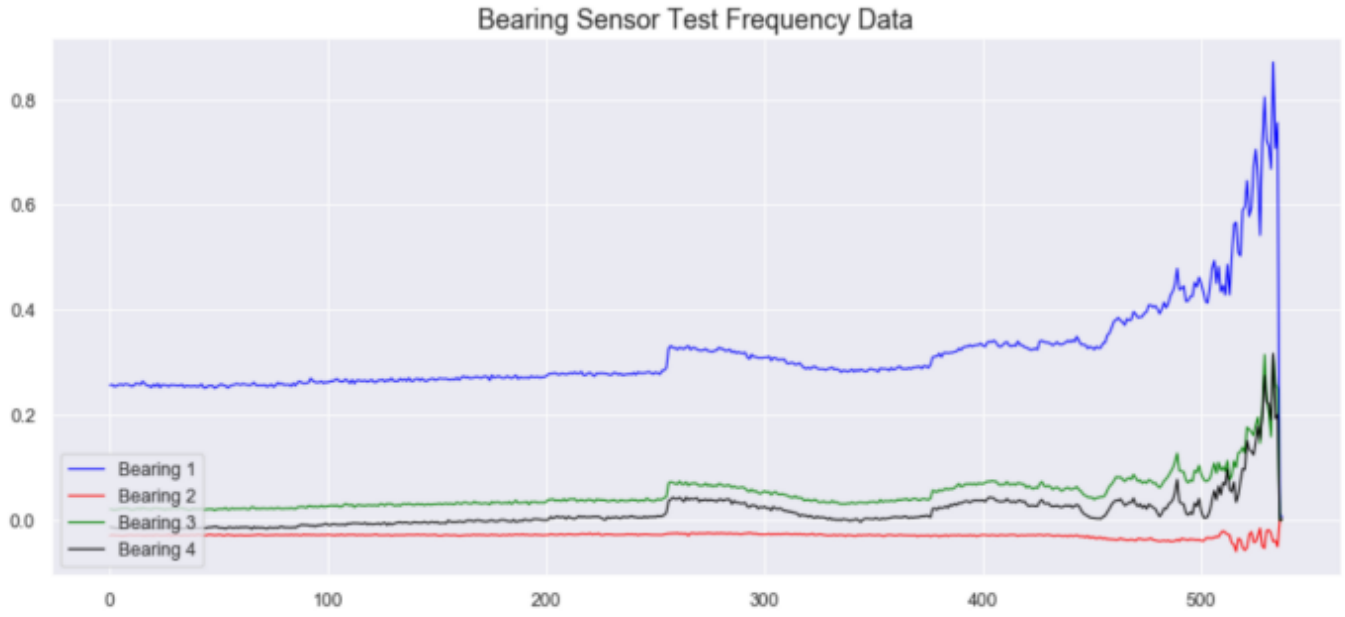
Let's first look at the training data in the frequency domain.

```
# frequencies of the healthy sensor signal
fig, ax = plt.subplots(figsize=(14, 6), dpi=80)
ax.plot(train_fft[:,0].real, label='Bearing 1', color='blue', animated = True, linewidth=1)
ax.plot(train_fft[:,1].imag, label='Bearing 2', color='red', animated = True, linewidth=1)
ax.plot(train_fft[:,2].real, label='Bearing 3', color='green', animated = True, linewidth=1)
ax.plot(train_fft[:,3].real, label='Bearing 4', color='black', animated = True, linewidth=1)
plt.legend(loc='lower left')
ax.set_title('Bearing Sensor Training Frequency Data', fontsize=16)
plt.show()
```



There is nothing notable about the normal operational sensor readings. Now, let's look at the sensor frequency readings leading up to the bearing failure.

```
# frequencies of the degrading sensor signal
fig, ax = plt.subplots(figsize=(14, 6), dpi=80)
ax.plot(test_fft[:,0].real, label='Bearing 1', color='blue', animated = True, linewidth=1)
ax.plot(test_fft[:,1].imag, label='Bearing 2', color='red', animated = True, linewidth=1)
ax.plot(test_fft[:,2].real, label='Bearing 3', color='green', animated = True, linewidth=1)
ax.plot(test_fft[:,3].real, label='Bearing 4', color='black', animated = True, linewidth=1)
plt.legend(loc='lower left')
ax.set_title('Bearing Sensor Test Frequency Data', fontsize=16)
plt.show()
```



We can clearly see an increase in the frequency amplitude and energy in the system leading up to the bearing failures.

To complete the pre-processing of our data, we will first normalize it to a range between 0 and 1. Then we reshape our data into a format suitable for input into an LSTM network. LSTM cells expect a 3 dimensional tensor of the form [data samples, time steps, features]. Here, each sample input into the LSTM network represents one step in time and contains 4 features — the sensor readings for the four bearings at that time step.

```
# normalize the data
scaler = MinMaxScaler()
X_train = scaler.fit_transform(train)
X_test = scaler.transform(test)
scaler_filename = "scaler_data"
joblib.dump(scaler, scaler_filename)
```

```
['scaler_data']
```

```
# reshape inputs for LSTM [samples, timesteps, features]
X_train = X_train.reshape(X_train.shape[0], 1, X_train.shape[1])
print("Training data shape:", X_train.shape)
X_test = X_test.reshape(X_test.shape[0], 1, X_test.shape[1])
print("Test data shape:", X_test.shape)
```

```
Training data shape: (445, 1, 4)
Test data shape: (538, 1, 4)
```

One of the advantages of using LSTM cells is the ability to include multivariate features in your analysis. Here, it's the four sensor readings per time step. However, in an online fraud anomaly detection analysis, it could be features such as the time of day, dollar amount, item purchased, internet IP per time step.

Neural Network Model

We will use an autoencoder neural network architecture for our anomaly detection model. The autoencoder architecture essentially learns an "identity" function. It will take the input data, create a compressed representation of the core / primary driving features of that data and then learn to reconstruct it again. For instance, input an image of a dog, it will compress that data down to the core constituents that make up the dog picture and then learn to recreate the original picture from the compressed version of the data.

The rationale for using this architecture for anomaly detection is that we train the model on the "normal" data and determine the resulting reconstruction error. Then, when the model encounters data that is outside the norm and attempts to reconstruct it, we will see an increase in the reconstruction error as the model was never trained to accurately recreate items from outside the norm.

We create our autoencoder neural network model as a Python function using the Keras library.

```
# define the autoencoder network model
def autoencoder_model(X):
    inputs = Input(shape=(X.shape[1], X.shape[2]))
    L1 = LSTM(16, activation='relu', return_sequences=True,
              kernel_regularizer=regularizers.l2(0.00))(inputs)
    L2 = LSTM(4, activation='relu', return_sequences=False)(L1)
    L3 = RepeatVector(X.shape[1])(L2)
    L4 = LSTM(4, activation='relu', return_sequences=True)(L3)
    L5 = LSTM(16, activation='relu', return_sequences=True)(L4)
    output = TimeDistributed(Dense(X.shape[2]))(L5)
    model = Model(inputs=inputs, outputs=output)
    return model
```

In the LSTM autoencoder network architecture, the first couple of neural network layers create the compressed representation of the input data, the encoder. We then use a repeat vector layer to distribute the compressed representational vector across the time steps of the decoder. The final output layer of the decoder provides us the reconstructed input data.

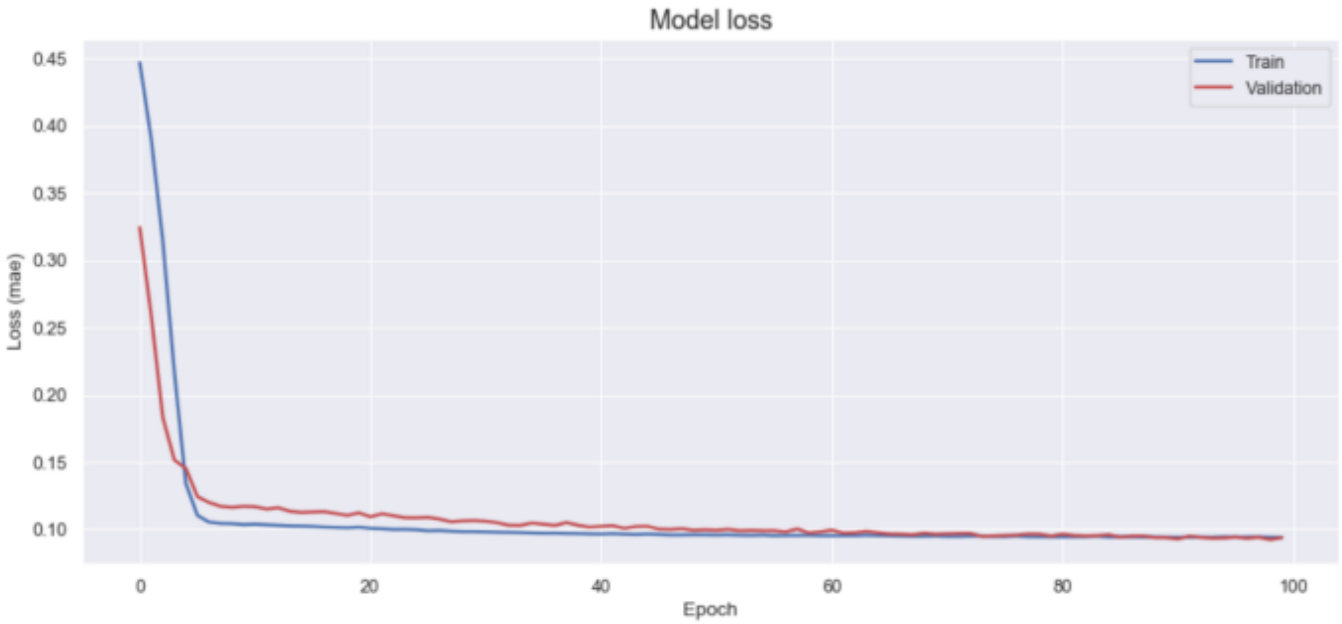
We then instantiate the model and compile it using Adam as our neural network optimizer and mean absolute error for calculating our loss function.

```
# create the autoencoder model
model = autoencoder_model(X_train)
model.compile(optimizer='adam', loss='mae')
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 1, 4)              0
_________________________________________________________________
lstm_1 (LSTM)                (None, 1, 16)             1344
_________________________________________________________________
lstm_2 (LSTM)                (None, 4)                 336
_________________________________________________________________
repeat_vector_1 (RepeatVecto (None, 1, 4)              0
_________________________________________________________________
lstm_3 (LSTM)                (None, 1, 4)              144
_________________________________________________________________
lstm_4 (LSTM)                (None, 1, 16)             1344
_________________________________________________________________
time_distributed_1 (TimeDist (None, 1, 4)              68
=================================================================
Total params: 3,236
Trainable params: 3,236
Non-trainable params: 0
```

```
# fit the model to the data
nb_epochs = 100
batch_size = 10
history = model.fit(X_train, X_train, epochs=nb_epochs, batch_size=batch_size,
                    validation_split=0.05).history
```

```
# plot the training losses
fig, ax = plt.subplots(figsize=(14, 6), dpi=80)
ax.plot(history['loss'], 'b', label='Train', linewidth=2)
ax.plot(history['val_loss'], 'r', label='Validation', linewidth=2)
ax.set_title('Model loss', fontsize=16)
ax.set_ylabel('Loss (mae)')
ax.set_xlabel('Epoch')
ax.legend(loc='upper right')
plt.show()
```



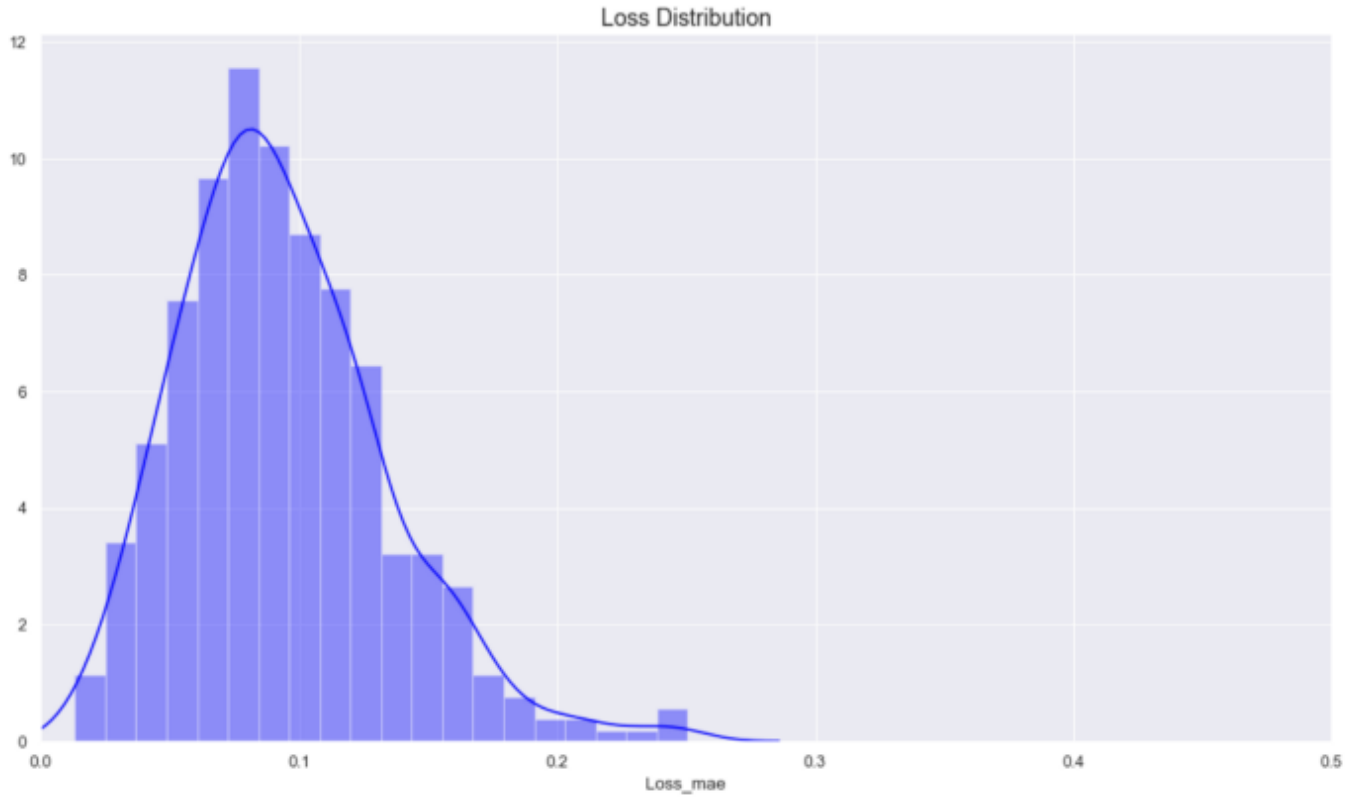
Loss Distribution

By plotting the distribution of the calculated loss in the training set, we can determine a suitable threshold value for identifying an anomaly. In doing this, one can make sure that this threshold is set above the “noise level” so that false positives are not triggered.

```
# plot the loss distribution of the training set
X_pred = model.predict(X_train)
X_pred = X_pred.reshape(X_pred.shape[0], X_pred.shape[2])
X_pred = pd.DataFrame(X_pred, columns=train.columns)
X_pred.index = train.index

scored = pd.DataFrame(index=train.index)
Xtrain = X_train.reshape(X_train.shape[0], X_train.shape[2])
scored['Loss_mae'] = np.mean(np.abs(X_pred-Xtrain), axis = 1)
plt.figure(figsize=(16,9), dpi=80)
plt.title('Loss Distribution', fontsize=16)
sns.distplot(scored['Loss_mae'], bins = 20, kde= True, color = 'blue');
plt.xlim([0.0,.5])
```

```
(0.0, 0.5)
```



Based on the above loss distribution, let's try a threshold value of 0.275 for flagging an anomaly. We then calculate the reconstruction loss in the training and test sets to determine when the sensor readings cross the anomaly threshold.

```
# calculate the loss on the test set
X_pred = model.predict(X_test)
X_pred = X_pred.reshape(X_pred.shape[0], X_pred.shape[2])
X_pred = pd.DataFrame(X_pred, columns=test.columns)
X_pred.index = test.index
```

```
# calculate the same metrics for the training set
# and merge all data in a single dataframe
for plotting
X_pred_train = model.predict(X_train)
X_pred_train = X_pred_train.reshape(X_pred_train.shape[0], X_pred_train.shape[2])
X_pred_train = pd.DataFrame(X_pred_train, columns=train.columns)
X_pred_train.index = train.index

scored_train = pd.DataFrame(index=train.index)
scored_train['Loss_mae'] = np.mean(np.abs(X_pred_train-Xtrain), axis = 1)
scored_train['Threshold'] = 0.275
scored_train['Anomaly'] = scored_train['Loss_mae'] > scored_train['Threshold']
scored = pd.concat([scored_train, scored])
```

Note that we've merged everything into one dataframe to visualize the results over time. The red line indicates our threshold value of 0.275.

```
# plot bearing failure time plot
scored.plot(logy=True,  figsize=(16,9), ylim=[1e-2,1e2], color=['blue','red'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a2eceef98>
```

Our neural network anomaly analysis is able to flag the upcoming bearing malfunction well in advance of the actual physical bearing failure by detecting when the sensor readings begin to diverge from normal operational values.

```
# save all model information, including weights, in h5 format
model.save("Cloud_model.h5")
print("Model saved")
```

```
Model saved
```

Update:

In the next article, we'll deploy our trained AI model as a REST API using Docker and Kubernetes for exposing it as a service.