AutoEncoder for Interpolation

Rahul Bhadani*

05 January 2021

Abstract

In physical science, sensor data are collected over time to produce timeseries data. However, depending on the real-world condition and underlying physics of the sensor, data might be noisy. Besides, the limitation of sample-time on sensors may not allow collecting data over all the timepoints, may require some form of interpolation. Interpolation may not be smooth enough, fail to denoise data, and derivative operation on noisy sensor data may be poor that do not reveal any high order dynamics. In this article, we propose to use Autoencoder to perform interpolation that also denoise data simultaneously. A brief example using a real-world is also provided.

1 Autoencoder

Autoencoders [Ballard, 1987] are another Neural Network used to reproduce the inputs in a compressed fashion. Autoencoder has a special property in which the number of input neurons is the same as the number of output neurons. See the Figure 1. The goal of Autoencoder is to create a representation of the input at the output layer such that both output and input are similar but the actual use of the Autoencoder is for determining a compressed version of the input data with the lowest amount of loss in data. This is very similar to what Principal Component Analysis does, in a black-box manner. Encoder part of Autoencoder compresses the data at the same time ensuring that the important data is not lost but the size of the data is reduced.

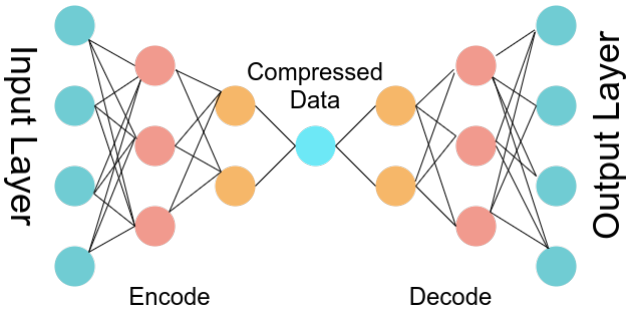

Figure 1: A Schematic of Autoencoder drawn using draw.io

The downside of using Autoencoder for interpolation is that the compressed data is a black box representation—we do not know the structure of the data in the compressed version. Suppose we have a dataset with 10 parameters and we train an autoencoder over this data. The encoder does not omit some of the parameters for better representation but it fuses the parameters to create a compressed version but with fewer parameters (brings the number of parameters down to, say, 5 from 10). Autoencoder has two parts, encoder, and decoder. The encoder compresses the input data and the decoder does the opposite to produce the uncompressed version of the data to produce a reconstructed input as close to the original one as possible.

*The University of Arizona, rahulbhadani@email.arizona.edu

2 Interpolation Method

Interpolation is a process of guessing the value of a function between two data points. For example, you are given $x = [1, 3, 5, 7, 9]$, and $y = [230.02, 321.01, 305.00, 245.75, 345.62]$, and based on the given data you want to know the value of y given $x = 4$. There are plenty of interpolation methods available in the literature — some model-based and some are model-free, i.e. data-driven. The most common way of achieving interpolation is through data-fitting. As an example, you use linear regression analysis to fit a linear model to the given data. In linear regression [Kutner et al., 2005], given the explanatory/predictor variable, X, and the response variable, Y, the data is fitted using the formula $Y = \beta_0 + \beta_1 X$ where β_0 and β_1 are determined using least square fit. As the name suggests, linear regression is linear, i.e., it fits a straight line even though the relationship between predictor and response variable might be non-linear. However, the most general form of interpolation is polynomial fitting. Given k sample points, it is straightforward to fit a polynomial of degree $k-1$. Given the data set $\{x_i, y_i\}$, the polynomial fitting is obtained by determining polynomial coefficients a_i of function

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{k-1} x^{k-1}$$

by solving matrix inversion from the following expression:

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{k-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{k-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_1 & x_1^2 & \cdots & x_1^{k-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{k-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{k-1} \end{pmatrix}$$

Once we have coefficients a_i, we can find the value of function f for any x. There are some specific cases of polynomial fitting where a piecewise cubic polynomial is fitted to data. A few other non-parametric methods include cubic spline, smoothing splines, regression splines, kernel regression, and density estimation [Hastie et al., 2009] (Figure 2).

However, the point of this article is not polynomial fitting, but rather interpolation. Polynomial fitting just happens to facilitate interpolation. However, there is an issue with polynomial fitting methods — whether it is parametric or non-parametric, they behave the way they are taught. What it means is that if data is clean, the fitting will be clean and smooth, but if data is noisy, the fitting will be noisy. This issue is more prevalent in sensor data, for example, hear-beat data captured from your heart-rate sensor, distance data from LiDAR, CAN Bus speed data from your car, GPS data, etc.

Further, because of the noise, they are harder to deal with, especially if your algorithm requires performing double, or second derivative on such data. In general, those sensor data are timeseries data, i.e. they are collected over time, thus the response variable might be some physical quantity such as speed, the distance of objects from LiDAR mounted on the top of a self-driving car, heart-rate, and predictor variable is time. While operating on such data, there can be a few objectives: we want to have data interpolated to some time-stamp over which our sensor couldn't record any response, but since sensors operate in the real-time world and because of the underlying physics, those data stay noisy, we also want a reliable interpolation that is not impacted by sensor noise. Further, our requirement may also include derivatives of such timeseries data. Derivatives tend to amplify the noise present in the underlying timeseries data [Bhadani et al., 2019]. What if there is a way by which we can get an underlying representation of the data, discarding the noise at the same time? Autoencoder comes to the rescue to achieve our objective in such a case.

3 Autoencoder as Interpolator

To demonstrate the denoising + interpolation objective using Autoencoder, we use an example of distance data collected from a vehicle by our lab, where the response variable is the distance of the vehicle ahead of

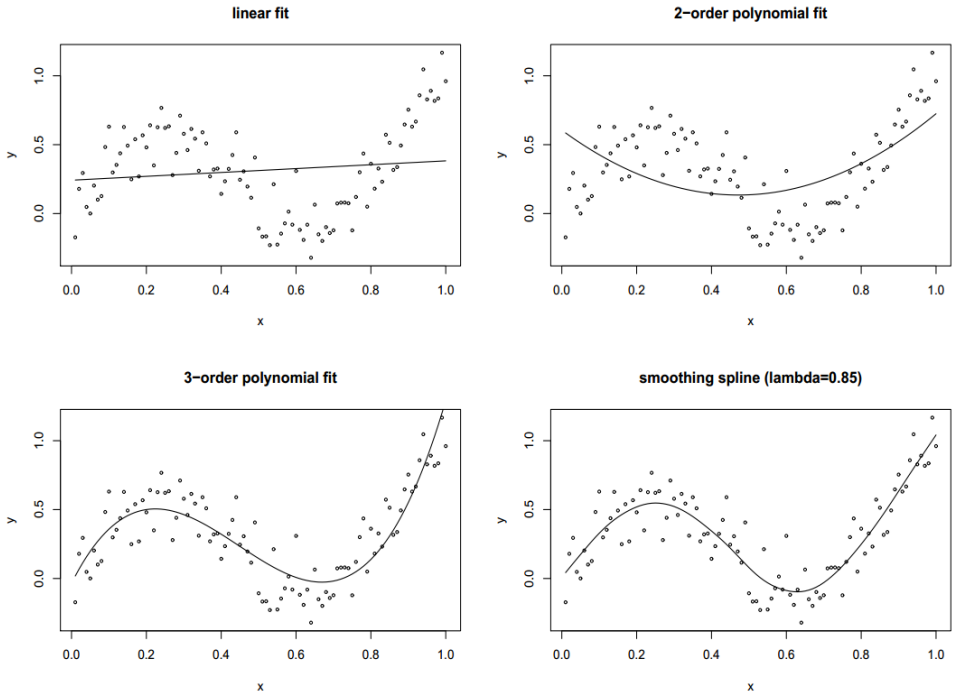

Figure 2: Examples of data-fitting.

our vehicle, and the predictor is time. We have made a small subset of the data available on the GitHub repo at https://github.com/rahulbhadani/medium.com/blob/master/data/lead_dist_sample.csv as a part of the demonstration that you are free to use. However, it is really small and serves no purpose beyond the tutorial described in this article.

Note: *Before you use data, we should point out that the time (predictor) and message (response) must be re-scaled. In our case, the original time starts from 1594247088.289515 (in POSIX format, in seconds) and ends at 1594247110.290019. We normalized the time value using the formula* `(time - start_time)/(end_time - start_time)`*. Similarly, the response variable was normalized using* `(message - message_min)/(message_max -message_min)`*. The sample data provided in the GitHub repo is already normalized and you can reuse it out of the box.*

3.1 Training

```
import glob
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
df = pd.read_csv("../data/lead_dist_sample.csv")
time = df['Time']
message = df['Message']
import tensorflow as tf
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(units = 1, activation = 'linear', input_shape=[1]))
model.add(tf.keras.layers.Dense(units = 128, activation = 'relu'))
model.add(tf.keras.layers.Dense(units = 64, activation = 'relu'))
```

```
model.add(tf.keras.layers.Dense(units = 32, activation = 'relu'))
model.add(tf.keras.layers.Dense(units = 64, activation = 'relu'))
model.add(tf.keras.layers.Dense(units = 128, activation = 'relu'))
model.add(tf.keras.layers.Dense(units = 1, activation = 'linear'))
model.compile(loss='mse', optimizer="adam")
model.summary()
# Training
model.fit( time, message, epochs=1000, verbose=True)
```

As you can see, We have not performed any regularization as we deliberately want to do overfitting so that we can use the underlying nature of data to the full extent. Now it's time to make a prediction. You will see that we rescaled back the time axis to original values before making predictions. For this example, we had `time_original[0] = 1594247088.289515` ,`time_original[-1] = 1594247110.290019` , `msg_min = 33`, `msg_max = 112`.

```
newtimepoints_scaled = np.linspace(time[0] - (time[1] - time[0]),time[-1], 10000)
y_predicted_scaled = model.predict(newtimepoints_scaled)
newtimepoints =
    newtimepoints_scaled*(time_original[-1] - time_original[0]) + time_original[0]
y_predicted = y_predicted_scaled*(msg_max - msg_min) + msg_min
```

Note that we are creating much denser time-points in variable `newtimepoints_scaled` which allows me to interpolate data on unseen time-points. Finally, the is the curve comparing interpolated data and original data is shown in Figure 3.

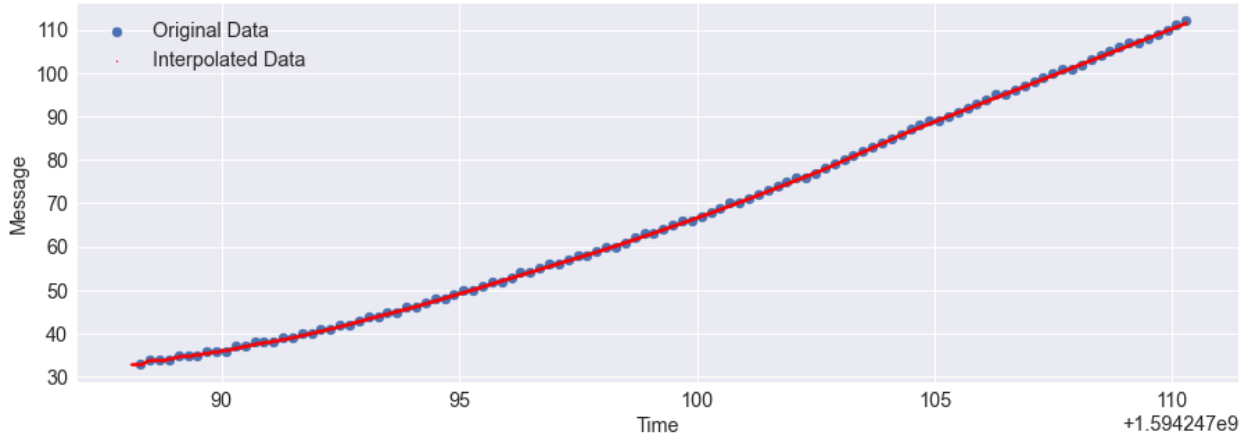

Figure 3: Interpolated Data and Original Data

4 Concluding Remarks

While we trained for only 1000 epochs, your training might not be that short, if your data is big. The biggest advantage of this method is taking derivatives, as from Figure 4, it is clear that the derivative performed on the original data is poor — may not even represent the true derivative!

```
df_interpolation = pd.DataFrame()
df_interpolation['Time'] = newtimepoints
df_interpolation['Message'] = y_predicted
df_interpolation['diff'] =
    df_interpolation['Message'].diff()/df_interpolation['Time'].diff()
```

```
df_original = pd.DataFrame()
df_original['Time'] = time*(1594247110.290019 - 1594247088.289515) + 1594247088.289515
df_original['Message']  = message*(112 - 33) + 33
df_original['diff'] = df_original['Message'].diff()/df_original['Time'].diff()
# Display the result
import matplotlib.pylab as pylab
params = {'legend.fontsize': 'x-large',
          'figure.figsize': (15, 5),
         'axes.labelsize': 'x-large',
         'axes.titlesize':'x-large',
         'xtick.labelsize':'x-large',
         'ytick.labelsize':'x-large'}
pylab.rcParams.update(params)
plt.scatter(df_original['Time'], df_original['diff'], label='Derivative on Original Data')
plt.scatter(df_interpolation['Time'], df_interpolation['diff'],
    s= 10, c = 'red', label='Derivative on Interpolated Data')
plt.xlabel('Time')
plt.ylabel('Message')
plt.legend()
plt.show()
```

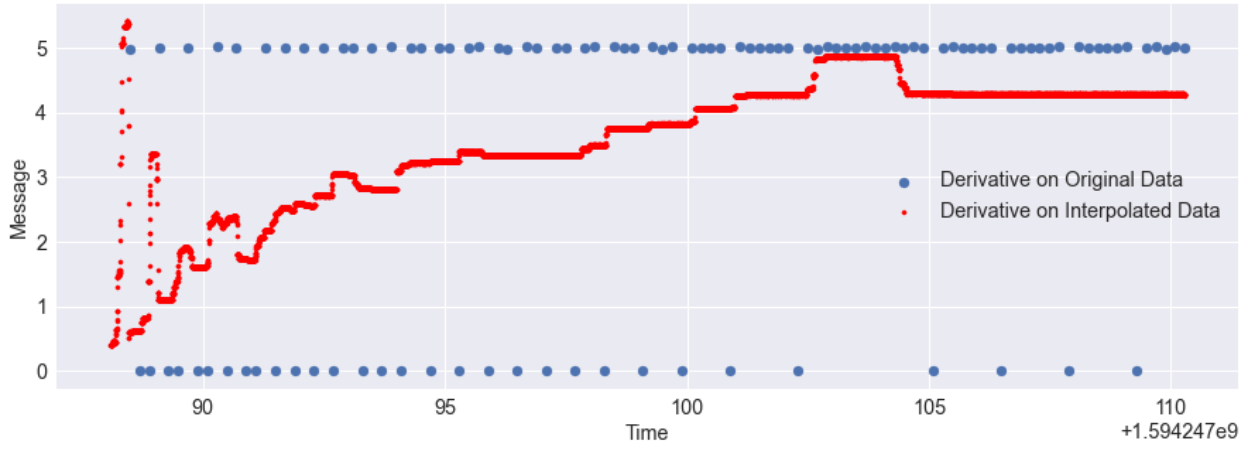

Figure 4: A comparison of calculating derivative on original and interpolated data

5 Acknowledgement

Originally, this article was produced by the author in Medium.com at https://rahulbhadani.medium.com/tensorflow-2-how-to-use-autoencoder-for-interpolation-91fefd4516c9.

References

Dana H Ballard. Modular learning in neural networks. In *AAAI*, pages 279–284, 1987.

Rahul Bhadani, Matthew Bunting, Benjamin Seibold, Raphael Stern, Shumo Cui, Jonathan Sprinkle, Benedetto Piccoli, and Daniel B Work. Real-time distance estimation and filtering of vehicle headways for smoothing of traffic waves. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 280–290, 2019.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

Michael H Kutner, Christopher J Nachtsheim, John Neter, William Li, et al. *Applied linear statistical models*, volume 5. McGraw-Hill Irwin New York, 2005.