Polynomial Curve Fitting

***Abstract*—This document contains theory behind curve fitting model**

1 Objective

Our objective is to implement the best fit line polynomial Curve.

2 Load Dataset

Create a sinusoidal data function

$$y = A\sin 2\pi ft + n(t) \tag{2.0.1}$$

with random noise included in the target values training set comprising N observations of t, written

$$t = \begin{pmatrix} t_1 & , & . & . & t_N \end{pmatrix}^T \tag{2.0.2}$$

together with corresponding observations of the values of y, denoted

$$y = \begin{pmatrix} y_1 & , & . & . & y_N \end{pmatrix}^T \tag{2.0.3}$$

Fig 0 was generated by choosing values of t_n, for n

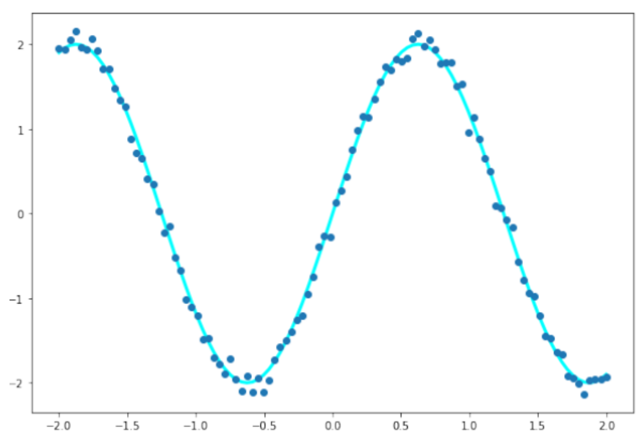

Fig. 0: Sinusoidal Dataset

= 1, . . . , N, where N= 100 and spaced uniformly. Data set y was obtained by first computing the corresponding values of the function $A\sin 2\pi ft$ and then adding a small level of random noise having a random distribution to each such point in order to obtain the corresponding value.

```
# Data Creation
def create_data():
    t = PolynomialFeatures(degree=6).
        fit_transform(np.linspace(−2,2,100).
        reshape(100,−1))
    t[:,1:] = MinMaxScaler(feature_range=(−2,2),
        copy=False).fit_transform(t[:,1:])
    l = lambda t_i: 2∗np.sin(0.8∗np.pi∗t_i)
    data = l(t[:,1])
    noise = np.random.normal(0,0.1,size=np.shape
        (data))
    y = data+noise
    y= y.reshape(100,1)
    return {'t':t,'y':y}
```

3 Polynomial Curve Fitting

To find a line that best resembles the underlying pattern of the training data shown in the graph. By using the least squares method followed by Stochastic gradient descent to corresponding estimated responses,The objective function to be minimized is

$$Q(w) = \sum_{i=1}^{n} Q_i(w) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{3.0.1}$$

$$\implies \sum_{i=1}^{n} (w_1 + w_2 t_i - y_i)^2 \tag{3.0.2}$$

The Last Line in the above pseudocode for this specific problem will become,

(3.0.3)

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} := \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} - \eta \begin{pmatrix} \frac{\partial}{\partial w_1}(w_1 + w_2 t_i - y_i)^2 \\ \frac{\partial}{\partial w_2}(w_1 + w_2 t_i - y_i)^2 \end{pmatrix} \tag{3.0.4}$$

$$= \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} - \eta \begin{pmatrix} 2(w_1 + w_2 t_i - y_i) \\ 2t_i(w_1 + w_2 t_i - y_i) \end{pmatrix} \tag{3.0.5}$$

Note that in each iteration (also called update),only the gradient evaluated at a single point t_i instead of evaluating at the set of all samples.

```
def batch_gradient_descent(t,y,w,eta):
```

```
    derivative = np.sum([−(y[d]−np.dot(w.T.copy
        (),t[d,:]))∗t[d,:].reshape(np.shape(w)) for d
         in range(len(t))],axis=0)
    return eta∗(1/len(t))∗derivative

# Update w
w_s = []
Error = []
for i in range(iterations):
    # Calculate error
    error = (1/2)∗np.sum([(y[i]−np.dot(w.T,t[i,:]))
        ∗∗2 for i in range(len(t))])
    Error.append(error)
    w −= batch_gradient_descent(t,y,w,eta)
\end{listing}
Code to initialize variables.\\
\begin{lstlisting}
# initialize variables
data = create_data()
t = data['t']
y = data['y']
w = np.random.normal(size=(np.shape(t)[1],1))
eta = 0.1
iterations = 10000
batch = 10
```

Code to initialize variables.

```
# initialize variables
data = create_data()
t = data['t']
y = data['y']
w = np.random.normal(size=(np.shape(t)[1],1))
eta = 0.1
iterations = 10000
batch = 10
```

Plotting the Predicted and actual plot

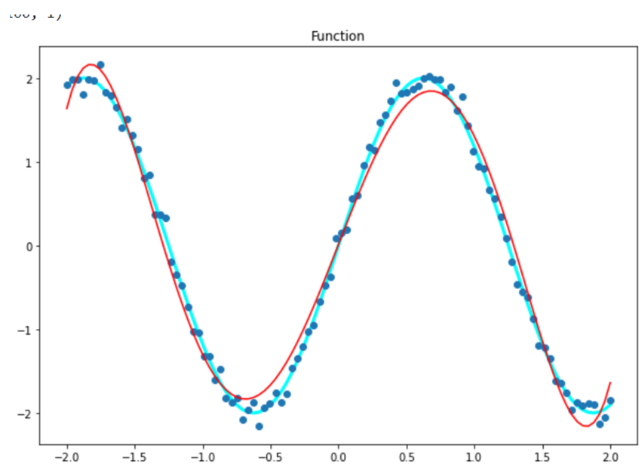

Fig. 0: Fitted Data

4 Observation

- Less oscillations and noisy steps taken towards the global minima of the loss function due to updating the parameters by computing the average of all the training samples rather than the value of a single sample.

Download Python codes from

```
https://github.com/ayushkesh/EE4015/tree/master/
    AI4
```