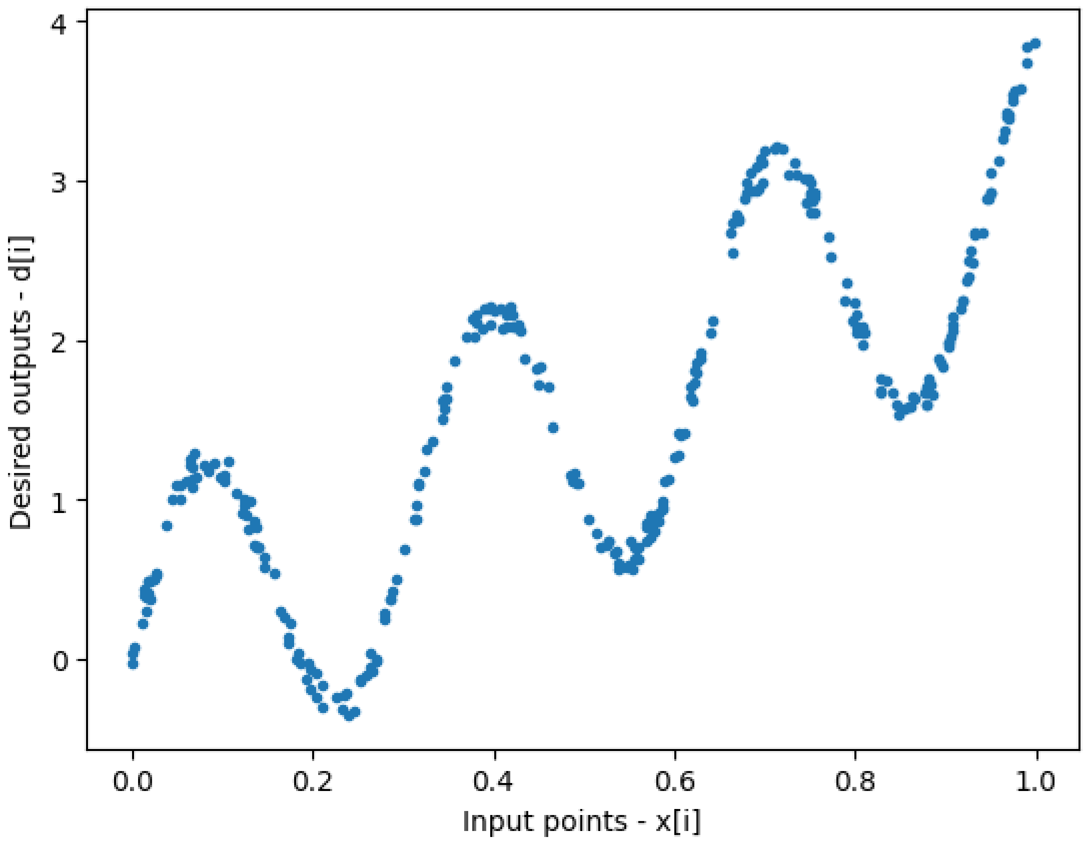
<u>Homework - 4</u>

No of training samples are 300, so n = 300
Input of the Neural Network is randomly uniform on [0,1], naming as x
There are n real numbers uniformly at random on [− 1 10 , 1 10], naming as v.
Desired output is calculated using : $sin(20x[i]) + 3x[i] + \nu[i], i = 1, ..., n.$

The plot of (xi , di), $i = 1, . . . , n$ is shown below



Considering $1\ X\ N\ X\ 1$ Neural Network. Let’s consider $1\ X\ N$ as first layer and $N\ X\ 1$ as a second layer.

Calculating weights for first layer $(1\ X\ N)$:
There are N weights from X input to N neurons and each neuron would be biased. So total weights for first layer is $2N$.

Calculating weights for first layer $(N\ X\ 1)$:
There are N weights from N neurons to 1 output neuron and it would be biased. So total weights for first layer is $N + 1$.

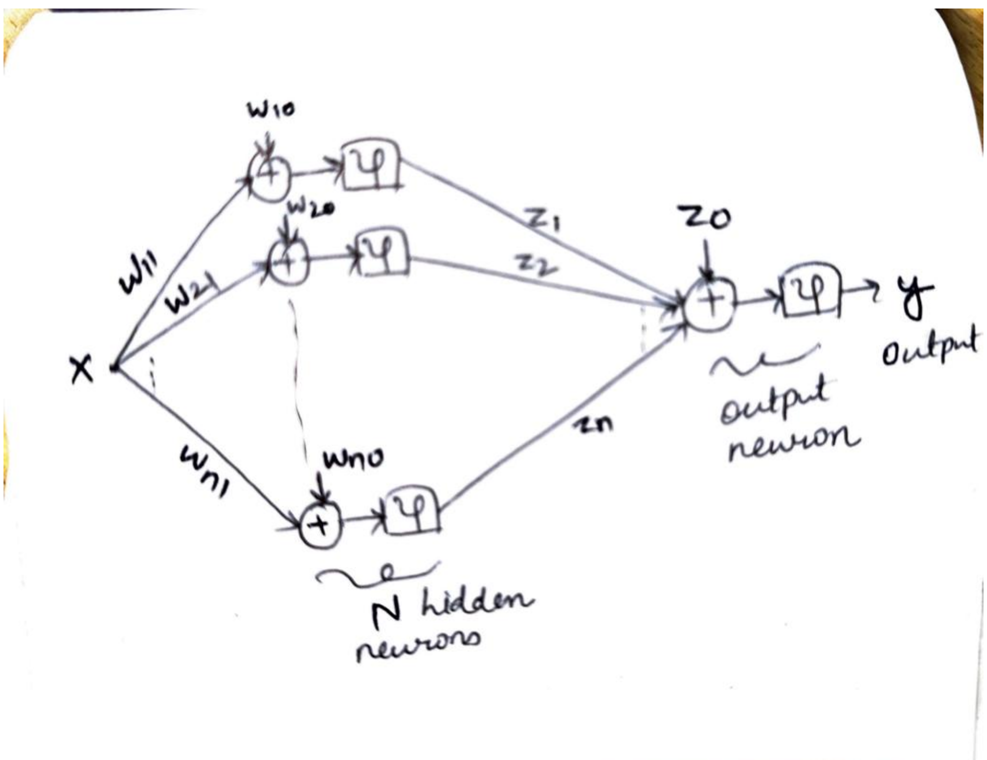
Hence, total weights of this $1\ X\ N\ X\ 1$ Neural Network is $3N + 1$.

Some initializations of the Neural Network :

N = 24 (hidden neurons)
η = 0.01
w_first_layer = np.array(np.random.uniform(-0.5,0.5, size = n))
w_first_layer_bias = np.array(np.random.uniform(-0.5,5, size = n))
w_second_layer = np.array(np.random.uniform(-0.5,5, size = n))
w_second_layer_bias = np.array(np.random.uniform(-0.5,5, size = 1))

Here, w represents weights, hence w_first_layer is weights from X input to N neurons in the first layer.

Please find the below diagram of $1\ X\ N\ X\ 1$ Neural Network



Here, w_{i1} is w_first_layer, $i = 1, \ldots, n$
w_{i0} is w_first_layer_bias, $i = 1, \ldots, n$
z_i is w_second_layer, $i = 1, \ldots, n$
z_0 is w_second_layer_bias

The output neuron has activation function φ(v) = v; all other neurons have activation function φ(v) = tanh v.

Pseudo Code for Backpropagation algorithm :

Initialization is already done for below variables in the above pages.
w_first_layer,w_first_layer_bias,w_second_layer,w_second_layer_bias,d,x,N,n,η

```
epoch = 0
m_s_e_array = [] // It will store MSE for all epochs

while(1):

        Initializing  sum_first_layer [n,N] = 0, activation_first_layer [n,N] = 0, sum_second_layer
        [n] = 0 and y [n] = 0

        // Doing forward propagation
        // Calculating local field for first layer
        for i = 1 to n:
                for j = 1 to N:
                        sum_first_layer[i][j] = (w_first_layer[j] * x[i][j]) + w_first_layer_bias[j]

        // Calculating the output of first layer from the activation function, here φ(v) = tanh v
        for i = 1 to n:
                for j = 1 to N:
                        activation_first_layer[i][j] = tanh(sum_first_layer[i][j])

        // Calculating local field for second layer
        for i = 1 to n:
                sum_second_layer[i] = 0
                for j = 1 to N:
                        sum_second_layer[i] +=(activation_first_layer[i][j] * w_second_layer[j]) +
                                w_second_layer_bias[j]

        // Calculating the output from the activation function, here φ(v) = v
        for i = 1 to n:
            y[i] = sum_second_layer[i]

        // Calculating the MSE for that epoch
        m_s_e = 0
        for i = 1 to n:
            m_s_e += (d[i] - y[i])^2
        m_s_e = m_s_e / n

        // Modifying η ← 0.9η whenever the MSE has increased
        if( m_s_e > m_s_e_array[epoch - 1]):
            η = η*0.9
```

```
// Storing MSE for all epochs
m_s_e_array.append(m_s_e)

// breaking the loop after MSE is less than or equal to threshold. Here threshold is 0.01
if(m_s_e <= 0.01):
    break

// Updating the epoch
epoch = epoch + 1

// Doing backward propagation and updating the weights

for i = 1 to n:
   // Calculating y with the given weights

   // Calculating local field for first layer
   for j = 1 to N:
        sum_first_layer[i][j] = (w_first_layer[j] * x[i][j]) + w_first_layer_bias[j]

  // Calculating the output of first layer from the activation function, here φ(v) = tanh v
  for j = 1 to N:
        activation_first_layer[i][j] = tanh(sum_first_layer[i][j])

  // Calculating local field for second layer
  sum_second_layer = 0
  for j = 1 to N:
      sum_second_layer +=(activation_first_layer[i][j] * w_second_layer[j]) +
        w_second_layer_bias[j]

  // Calculating the output from the activation function, here φ(v) = v
   y = sum_second_layer

  // Updating w_first_layer – gradient descent equation
  gradient_first_layer_weights[N] = 0
  for j = 1 to N:
       gradient_first_layer_weights[j] = x[i] * (d[i]-y) * (1 - (tanh((w_first_layer[j] * x[i][j]) +
           w_first_layer_bias[j] )^2 )* w_second_layer[j]

  for j = 1 to N:
     w_first_layer[j] = w_first_layer[j] + η*gradient_first_layer_weights[j]
```

```
// Updating w_first_layer_bias – gradient descent equation
gradient_first_layer_weights_bias[N] = 0
for j = 1 to N:
      gradient_first_layer_weights_bias[j] = 1 * (d[i]-y) * (1 - (tanh((w_first_layer[j] *
           x[i][j]) + w_first_layer_bias[j] )^2 )* w_second_layer[j]

for j = 1 to N:
   w_first_layer_bias[j] = w_first_layer_bias[j] + η*gradient_first_layer_weights_bias[j]

// Updating w_second_layer – gradient descent equation
gradient_second_layer_weights[N] = 0
for j = 1 to N:
      gradient_second_layer_weights[j] = (d[i]-y) * (tanh((w_first_layer[j] * x[i][j]) +
           w_first_layer_bias[j] )

for j = 1 to N:
   w_second_layer[j] = w_second_layer[j] + η*gradient_second_layer_weights[j]

// Updating w_second_layer_bias – gradient descent equation
gradient_second_layer_weights_bias = (d[i]-y) *1
w_second_layer_bias[j] = w_second_layer_bias[j] +
                η*gradient_second_layer_weights[j]
```

After the algorithm function is completed

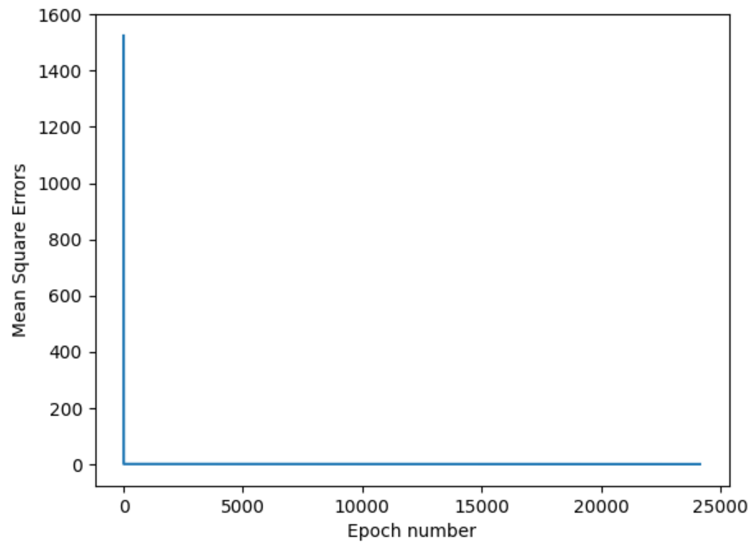
W_first_layer,w_first_layer_bias,w_second_layer,w_second_layer_bias are the final optimal weights

m_s_e_array would have the MSEs for each epoch, and size of m_s_e_array is the number of epochs required.

Gradient Descent Equations are :

```
for i = 1 to n:
        // Updating w_first_layer – gradient descent equation
        gradient_first_layer_weights[N] = 0
        for j = 1 to N:
            gradient_first_layer_weights[j] = x[i] * (d[i]-y) * (1 - (tanh((w_first_layer[j] * x[i][j]) +
                w_first_layer_bias[j] )^2 )* w_second_layer[j]

        for j = 1 to N:
          w_first_layer[j] = w_first_layer[j] + η*gradient_first_layer_weights[j]



        // Updating w_first_layer_bias – gradient descent equation
        gradient_first_layer_weights_bias[N] = 0
        for j = 1 to N:
            gradient_first_layer_weights_bias[j] = 1 * (d[i]-y) * (1 - (tanh((w_first_layer[j] *
                x[i][j]) + w_first_layer_bias[j] )^2 )* w_second_layer[j]

        for j = 1 to N:
          w_first_layer_bias[j] = w_first_layer_bias[j] + η*gradient_first_layer_weights_bias[j]

        // Updating w_second_layer – gradient descent equation
        gradient_second_layer_weights[N] = 0
        for j = 1 to N:
            gradient_second_layer_weights[j] = (d[i]-y) * (tanh((w_first_layer[j] * x[i][j]) +
                w_first_layer_bias[j] )

        for j = 1 to N:
          w_second_layer[j] = w_second_layer[j] + η*gradient_second_layer_weights[j]

        // Updating w_second_layer_bias – gradient descent equation
        gradient_second_layer_weights_bias = (d[i]-y) *1
        w_second_layer_bias[j] = w_second_layer_bias[j] +
                    η*gradient_second_layer_weights[j]
```

Graph for the number of epochs vs the MSE in the backpropagation algorithm

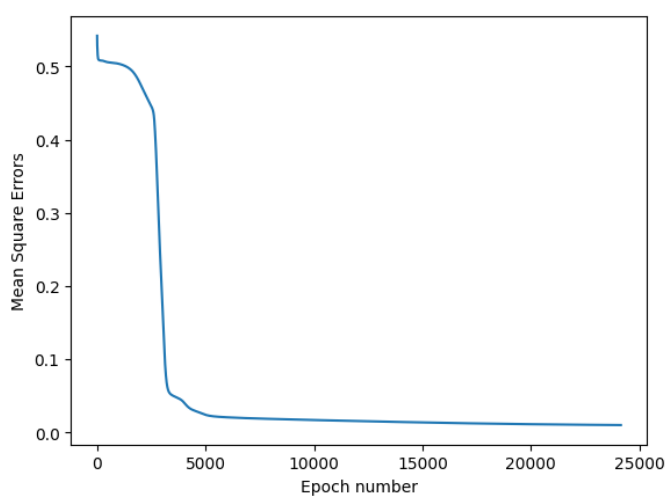


Here, approx 24147 epochs were used to converge MSE to 0.01. I can see that for initial epochs the MSE is very large and then it reduces.

Below is the array for MSE obtained :

```
1524.6108229296, 0.542018697263829, 0.5303619663186013, 0.5327218338390594
……., 0.010000610234480069, 0.010000397445203679, 0.010000184677296412,
0.009999719307483481]
```

As for the 0th epoch the MSE is very large compared to other epochs, the graph looks like a straight line but when zoomed in (after removing the 0th epoch) we can see that it is not a straight line.



Plot of desired output and actual output : $(xi\ ,\ di),\ i\ =\ 1, . . . , n.\ and\ (x,\ f(x,\ w0)\ ),\ x \in [0,\ 1]$