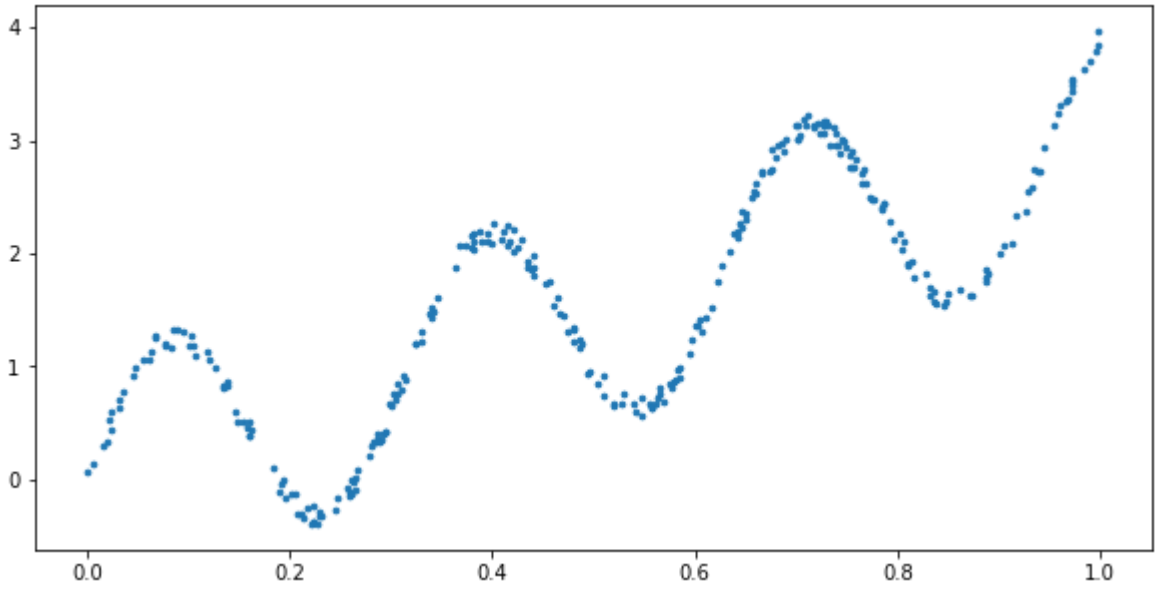
In [1]:
```
import numpy as np
from math import *
import matplotlib.pyplot as plt
```

Q 1 and 2

In [8]:
```
# Inputs
n= 300
x = [np.random.uniform(low =0, high = 1) for i in range(n)]
# Vector Fields
v = [np.random.uniform(low =-0.1, high = 0.1) for i in range(n)]
#v
mean = np.mean(x)
std  = np.std(x)
x_stand = []
for i in range(len(x)):
    x_stand.append([1,(x[i]-mean)/std])
```

Q 3

In [9]:
```
d = []
for i in range(0,n):
    d.append(sin(20*x[i]) + 3*x[i] + v[i])
d = np.asarray(d)
# d
fig, ax = plt.subplots(figsize = (10,5))
x = np.asarray(x)
#print(x)
plt.scatter(x,d, s  = 7)
plt.show()
```



In [250]:
```
import numpy as np
np.random.seed(100)

# hyperbolic Activation Function
def af(t):
    return tanh(t)

# Derivative of tanh(v)
def af_derivative(x):
    return 1-(tanh(x)**2)

def mse(x, d, w, nLayers, nNodes):
    c = 0
    n = len(x)
    for i in range(0,n):
        prev = x[i]
        for j in range(nLayers):
            u =[]
            p =[]
            for k in range(nNodes):
                t = np.dot(w[j]['weights'][k], prev)
                u.append(t)
            for l in u:
                p.append(af(l))
            p.append(1)
            prev = np.asarray(p)
        #print(w[nLayers]["weights"][0])
        y = np.dot(w[nLayers]["weights"][0], prev)
        c += (d[i] - y)**2
    return c/n

#np.random.seed(100)
class NeuralNetwork:

    def __init__(self,x=[],y=[],numLayers=2,numNodes=2, numOutputs = 1, eta=0.001,maxIter=10000, ep = 0):
        self.labels = y
        self.nLayers = numLayers
        self.nNodes = numNodes
        self.numOutputs = numOutputs
        self.eta = eta
        self.temp_eta = eta
        self.maxIt = maxIter
```

```
        self.ep = ep
        #self.train()
        #self.g = len(self.data[0])*numLayers + numLayers*numNodes + numNodes*numOutputs
#         newdata = []
#         for i in range(len(x)):
#             newdata.append(np.append(x[i],1))
        self.data = x
        self.weights = [{"weights":np.random.uniform(low =-2, high = 2, size = (self.nNodes, len(self.data[0
])))}]
        for i in range(self.nLayers):
            self.weights.append({"weights":np.random.uniform(low =-2, high = 2, size = (self.nNodes,self.nNod
es+1))})
        if self.nLayers >= 0:
            self.weights.append({"weights":np.random.uniform(low =-2, high = 2, size = (self.numOutputs,self.
nNodes+1))})
        self.temp_weights = self.weights


    def train(self):
        print(self.weights)
        temp_weights = self.weights
        temp_eta = self.temp_eta
        temp2 = mse(self.data, self.labels, self.weights, self.nLayers+1, self.nNodes)
        print(temp2)
        e = 0
        obj =[]
        epoch = []
        cos = 100000000
        while cos >= self.ep and e < self.maxIt:
            prev = cos
            for i in range(len(self.data)):
                self.backprop(self.labels[i], self.data[i])
                for m in range(len(self.weights)):
                    for j in range(len(self.weights[m]['weights'])):
                        for k in range(len(self.weights[m]['weights'][j])):
                            self.weights[m]['weights'][j][k] -= self.eta *self.weights[m]['g'][j][k]

            cos = mse(self.data, self.labels, self.weights, self.nLayers+1,self.nNodes)
#             epoch.append(e)
#             obj.append(cos)
#             e += 1

            if cos >= prev:
```

```
                self.eta = 0.9*self.eta
#                 obj =[]
#                 epoch = []
#                 e = 0
#                 obj.append(cos)
#                 epoch.append(e)
#                 cos = 100000000
#                 self.weights = temp_weights
#                 print(temp_weights)

                if self.eta <= 0.00001:
                    self.eta = temp_eta
#                     obj =[]
#                     epoch = []
#                     e = 0
#                     cos = 100000000
                    self.weights = [{"weights":np.random.uniform(low =-2, high = 2, size = (self.nNodes, len(
self.data[0])))}]
                    for i in range(self.nLayers):
                        self.weights.append({"weights":np.random.uniform(low =-2, high = 2, size = (self.nNod
es,self.nNodes+1))})
                    if self.nLayers >= 0:
                        self.weights.append({"weights":np.random.uniform(low =-2, high = 2, size = (self.numO
utputs,self.nNodes+1))})

            elif cos < prev:
                epoch.append(e)
                obj.append(cos)
                e += 1
        return self.weights, obj, epoch


    def feedforward(self,x=[]):
        prev = x
        r =[]
        r.append(prev)
        t =[]
        for j in range(self.nLayers+1):
            l = []
            s = []
            for m in range(self.nNodes):
                #print(self.weights[j]["weights"][m])
                s.append(np.matmul(self.weights[j]["weights"][m], prev))
```

```
            for k in s:
                l.append(af(k))
            l.append(1)
            prev = np.asarray(l)
            t.append(s)
            r.append(l)

        s =[]
        p = np.matmul(self.weights[self.nLayers+1]["weights"][0], l)
        s.append(p)
        t.append(s)
        return t,r, p

    def backprop(self, d, data):
        der = []
        #t,r= self.feedforward(data)
#         print('\n')
#         print("t = " + str(t))
#         print('\n')
#         print('r = ' + str(r))
        t,r,q = self.feedforward(data)

        for i in range(self.nLayers+1):
            a =[]
            for m in range(self.nNodes):
                a.append(af_derivative(t[i][m]))
            der.append(a)

        diff = []
        s =[]
        s.append(af_derivative(t[self.nLayers+1][0]))
        diff.append(d - q)
        der.append(s)

        for i in reversed(range(len(self.weights))):
            layer = self.weights[i]
            errors = []
            if i != len(self.weights)-1:
                for j in range(len(layer['weights'])):
                    error = 0

                    for k in range(len(self.weights[i + 1]['weights'])):
                        error += (self.weights[i + 1]['weights'][k][j] * self.weights[i + 1]['s'][k])
```

```
                    errors.append(error)
            else:
                errors.append(diff[0])
            layer['s'] = []
            for j in range(len(layer['weights'])):
                layer['s'].append(errors[j]*der[i][j])

# Finding the corresponding gradient vector

        for j in range(len(self.weights)):
            layer = self.weights[j]
            layer['g'] = []
            for k in range(len(self.weights[j]['weights'])):
                s =[]
                for m in range(len(self.weights[j]['weights'][k])):
                    s.append(((-(r[j][m])*self.weights[j]['s'][k])*2)/len(self.data))
                layer['g'].append(s)

        return 0.0
```

In [252]:
```
nHiddenLayers = 0
nNodes = 24
nOut = 1
# weights = weights_Initialization(x,nLayers,nNodes, nOut)
eta = 10
iter = 10000
ep = 0.005

w, obj ,epoch = NeuralNetwork(x_stand,d, nHiddenLayers, nNodes, nOut, eta, iter, ep).train()
```

```
[{'weights': array([[ 0.17361977, -0.88652246],
       [-0.30192964,  1.37910453],
       [-1.98112458, -1.51372352],
       [ 0.68299634,  1.30341102],
       [-1.45317364,  0.30037332],
       [ 1.56528782, -1.16319151],
       [-1.25868712, -1.56649244],
       [-1.12121003,  1.91449514],
       [ 1.2467326 , -1.31223595],
       [ 1.26489899, -0.90370501],
       [-0.27318327,  1.76011928],
       [ 1.27059752, -0.6555522 ],
       [-1.29835819, -0.50867181],
       [-1.97724597, -0.99029459],
       [ 1.18265003, -1.93898012],
       [ 0.39537351,  0.41521816],
       [-1.57940926, -0.47222622],
       [-1.85409577,  1.56164625],
       [ 1.92368343, -1.76023204],
       [ 1.56218378,  0.307606  ],
       [ 0.96991876,  0.52073575],
       [ 0.32736877, -1.91824347],
       [-1.15989369,  0.17873951],
       [ 1.07646068, -0.99721908]])}, {'weights': array([[-0.85641724,  1.40958035,  1.90002597,  1.53941317,
-0.56196862,
         0.39543578, -0.58081755, -0.63923914, -1.28767604, -1.04922317,
        -1.82055087,  0.02172572, -0.49499018,  0.3712216 ,  0.5197675 ,
        -1.42959874,  1.7353652 ,  1.78551952,  0.40918663, -0.44893488,
        -0.54724798, -1.18261889, -0.89293975, -1.01385648, -1.30556799]])}]
57.28091217862603
```

In [253]:
```
len(epoch)
```

Out[253]: 4006

In [254]:
```
cos = mse(x_stand, d, w, 1,24)
cos
```

Out[254]: 0.0049992814326523475

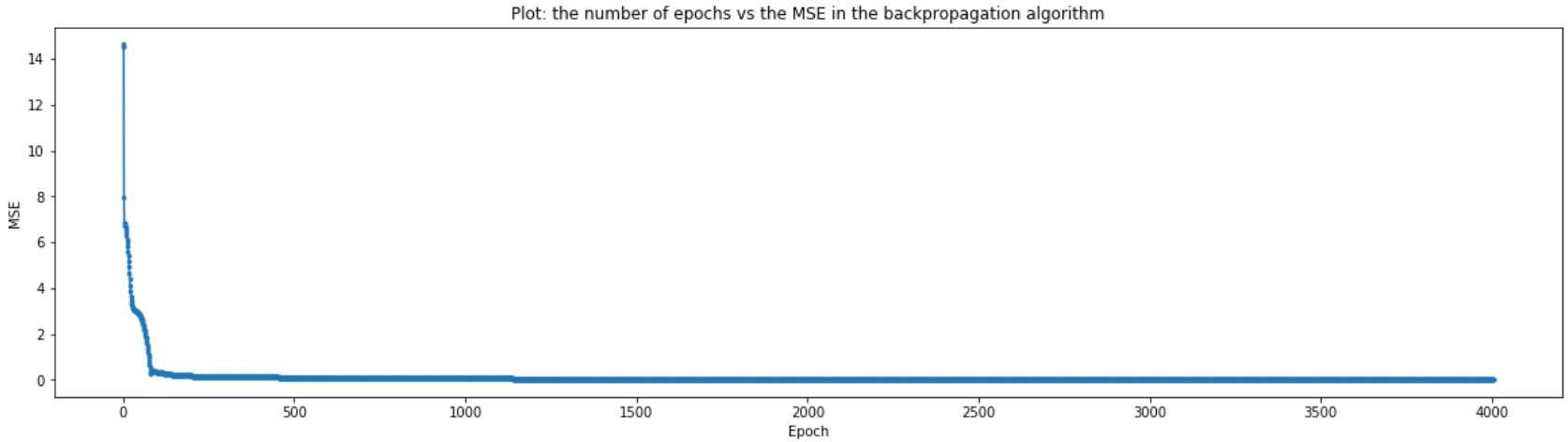
Q 4

In [255]:
```
def feedforward(data, weights, nLayers, nNodes):
    n =[]
    for i in range(len(data)):
        #print(i)
        prev = data[i]

        for j in range(nLayers):
            l = []
            s = []
            for m in range(nNodes):
                s.append(np.dot(weights[j]["weights"][m], prev))

            for k in s:
                l.append(tanh(k))
            prev = np.asarray(l)

        l.append(1)
        p = np.dot(weights[nLayers]["weights"][0], l)
        #print(p)

        n.append(p)
    return n
```

In [256]:
```
y = feedforward(x_stand, w, 1, 24)
```

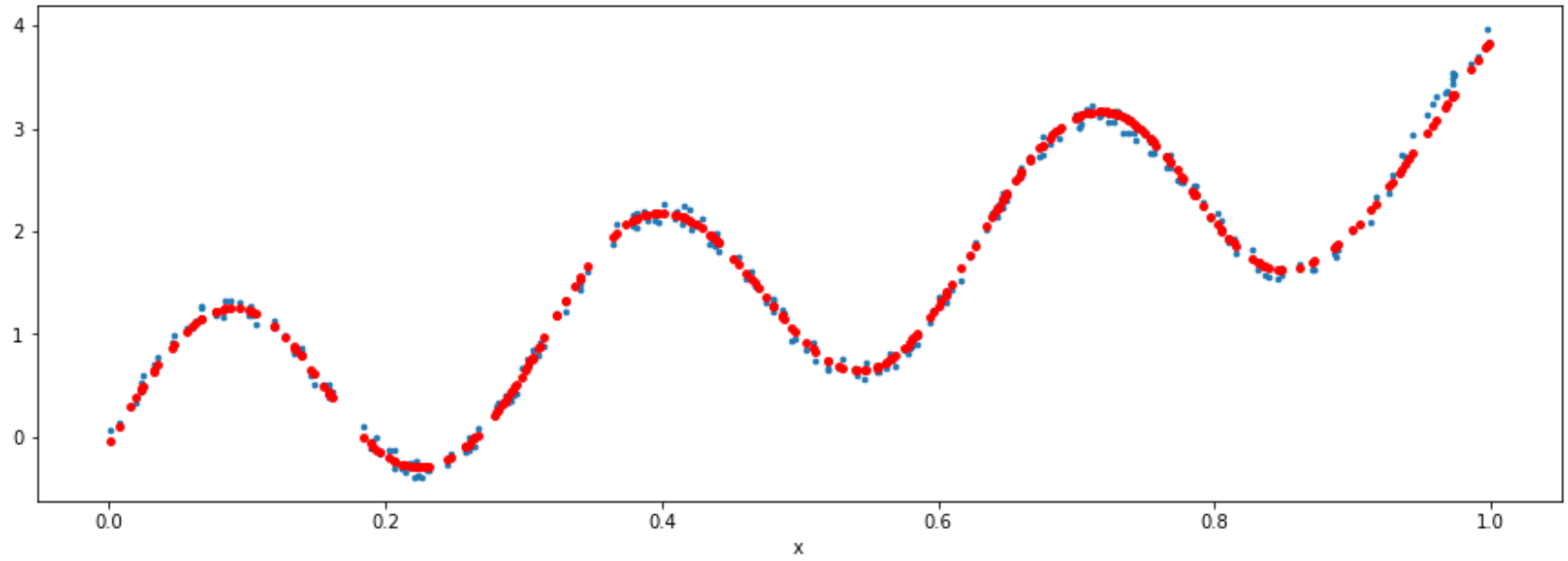
Plot: the number of epochs vs the MSE in the backpropagation algorithm.

In [257]:
```
fig, ax = plt.subplots(figsize = (20,5))
plt.scatter(epoch,obj, s  = 7)
plt.plot(epoch,obj)
plt.xlabel("Epoch")
plt.ylabel("MSE")
plt.title('Plot: the number of epochs vs the MSE in the backpropagation algorithm')
plt.show()
```



Q 5

In [258]:
```
fig, ax = plt.subplots(figsize = (15,5))
plt.scatter(x,d, s  = 7)
plt.scatter(x,y, c = 'r', s = 15)
plt.xlabel("x")
plt.show()
```



HW4 Pseudocode

1. For simplicity, I am using same number of neurons for all HIDDEN Layers
2. I am using a learning rate of 10
3. I standardized the data and appended the value 1 in every input vector to accommodate for the biases
4. I used the hyperbolic tangent activation function for every neuron except the output neuron.
5. **Activation Function:**

```
# hyperbolic Activation Function
def af(t):
    return tanh(t)
```

6. **Derivative of tanh(v)**

```
def af_derivative(x):
    return 1-(tanh(x)**2)
```

7. **mse Function:**

This is used to calculate the mean squared error corresponding to the given weights.

Algorithm:

```
Mse(data, weights):
    C = 0
    N = no of inputs
    For each input i in data:
        Previous = i
        For each layer “L”:
            initialize an empty list “u”
            initialize an empty list “p”
            for each node “n” in “L”:
                t = weights[L]['weights'][n] X Previous
                append t to the list “u”
            for each point “j” in the list “u”:
                append af(j) to the list “p”
            append 1 to the list “p”
            Previous = p
        y = weights[nLayers]['weights'][0] X Previous
        C = C + (d[i] - y)^2
        return c/N
```

HW4 Pseudocode

8. I created a **NeuralNetworks** class which has feedforward, backpropagation and train functions.
9. The NeuralNetworks class takes the data, labels, number of hidden layers, number of nodes per hidden layer, learning rate, and maximum iterations as input parameters.
10. **Function call:**

```
nLayers = 1
nNodes = 24
nOut = 1
eta = 10
iter = 100000
ep = 0.2
w, obj, epoch = NeuralNetwork(x, d, nLayers, nNodes, nOut, eta, iter, ep).train()
```

11. I used a list with name "**weights**" to store weights and gradient values. I am uniformly choosing weights between -1 and 1.
12. I used dictionaries of python with keys for each layer: "weights", "s", and "g" inside the list. The key "s" has the values before the weights of the layer in the feedback graph. The key "g" has the gradient descent values for all the weights in a layer.

```
weights = [{"weights":np.random.uniform(low =0, high = 1, size = (self.nNodes, len(self.data[0])))}]
for i in range(self.nLayers):
    self.weights.append({"weights":np.random.uniform(low =0, high = 1, size = (self.nNodes,self.nNodes+1))})          # 1 is for Biases
if self.nLayers >= 0:
    self.weights.append({"weights":np.random.uniform(low =0, high = 1, size = (self.numOutputs,self.nNodes+1))})      # 1 is for Biases
```

Ex: For number of hidden layers = 1 and num of nodes per hidden layer = 2.

```
[{'weights': array([[0.78888993, 0.36114814, 0.84949542],
                    [0.05266805, 0.89136156, 0.4386164 ]]),
          's': [-0.003484950437606789, -0.008284032450920822],
          'g': [[0.0017424752188033945, 0.0017424752188033945, 0.0017424752188033945],
                [0.004142016225460411,  0.004142016225460411, 0.004142016225460411]]},

{'weights': array([[0.22397066, 0.34915997, 0.12250444]]),
          's': [-0.14772231324694815],
          'g': [[0.06505347805517415, 0.05904723432620448, 0.07386115662347408]]}]
```

13. **Train function:**

HW4 Pseudocode

- This function uses the gradient descent vector from the backpropagation function to update the weights every epoch until maximum epochs are reached.
- If after an epoch, the mse is more than that of the mse of previous epoch, I am reducing the learning rate to 0.9 * learning rate
- If learning rate falls below 0.0001 then I am running the algorithm with new weights and original learning rate

Algorithm:

```
Train(self):
    temp_eta = self.temp_eta                                  # Temporarily storing the initial learning rate
    temp2 = mse(data, labels, weights, nLayers, nNodes)       # MSE with the initial weights
    print(temp2)
    e = 0
    initialize an empty list obj                              # List to store MSE after each epoch
    initialize an empty list epoch                            # List to store epochs
    cos = 100000000
    while cos >= ep and e <= maxIt:
      prev = cos
      for i in range(len(data)):
        self.backprop(labels[i], data[i])                     # Call for backpropagation function
        for m in range(len(weights)):
          for j in range(len(weights[m]['weights'])):
            for k in range(len(weights[m]['weights'][j])):
              weights[m]['weights'][j][k] -= self.eta * weights[m]['g'][j][k]     #Updation of weights

      cos = mse(data, self.labels, weights, nLayers,nNodes)
      if cos > prev:
        self.eta = 0.9*self.eta                               # Reducing the learning rate
        initialize an empty list obj
        initialize an empty list epoch
        e = 0
        cos = 100000000
        if self.eta <= 0.00001:                               # Starting from the beginning with different weights
          self.eta = temp_eta
```

```
            weights = [{"weights":np.random.rand(nNodes, len(data[0]))}]
            for i in range(nLayers-1):
               weights.append({"weights":np.random.rand(nNodes,nNodes+1)})
            if nLayers >= 0:
               weights.append({"weights":np.random.rand(numOutputs,nNodes+1)})
         elif cos <= prev:
            epoch.append(e)
            obj.append(cos)
            e += 1
      return weights, obj, epoch
```

14. Feedforward function:

For the given input, this records the local field values and values after the activation function of every neuron in each layer in the feedforward graph.

Algorithm:

```
   feedforward(self,x=[]):
      prev = x
      initialize an empty list “r”
      append “prev” to the list “r”
      initialize an empty list “t”
      for j in range(numLayers):
         initialize an empty list “l”
         initialize an empty list “s”
         for m in range(numNodes):
            append (weights[j]["weights"][m]) X prev to the list “s”
         for k in s:
            append af(k) to the list “l”
         append 1 to the list “l”          # This is for the bias of the neurons except the ones in the first layer
         prev = np.asarray(l)
         append “s” to the list “t”
         append “l” to the list “r”
```

```
initialize an empty list s
p = (weights[numLayers]["weights"][0]) X l
append “p” to the list “s”
append “s” to the list “t”
return t, r, p
```

15. Backpropagation function:

For the given input, this records the values before the weights of all the layers in the backward (feedback) graph and the corresponding gradient descent vector.

Algorithm:

```
Backpropagation (labels, data):
    initialize an empty list “der”
    t,r,q = self.feedforward(data)                    # Call for the feedforward function
    for i in range(numLayers):
        initialize an empty list “a”
        for m in range(numNodes):
            append af_derivative(t[i][m]) to the list “a”
        append “a” to the list “der”
    initialize an empty list “diff”
    initialize an empty list “s”
    append af_derivative(t[nLayers][0]) to the list “s”
    append d-q to the list “diff”
    append “s” to the list “der”

    for i in reversed(range(len(weights))):
        layer = weights[i]
        initialize an empty list “errors”
        if i != len(weights)-1:
            for j in range(len(layer['weights'])):
                error = 0
                for k in range(len(weights[i + 1]['weights'])):
```

```
                    error += (weights[i + 1]['weights'][k][j] * weights[i + 1]['s'][k])
                append error to the list “errors”
            else:
                append “diff[0]” to the list “errors”
            initialize an empty list “layer['s']”
            for j in range(len(layer['weights'])):
                append errors[j]*der[i][j] to the list “layer['s']”

    # Finding the corresponding gradient vector

        for j in range(len(weights)):
            layer = weights[j]
            initialize an empty list layer['g']
            for k in range(len(weights[j]['weights'])):
                initialize an empty list “s”
                for m in range(len(weights[j]['weights'][k])):
                    append ((-(r[j][m])*weights[j]['s'][k])*2)/len(data) to the list “s”          #Gradient values
                append “s” to the list “layer['g']”
        return 0.0
```

16.