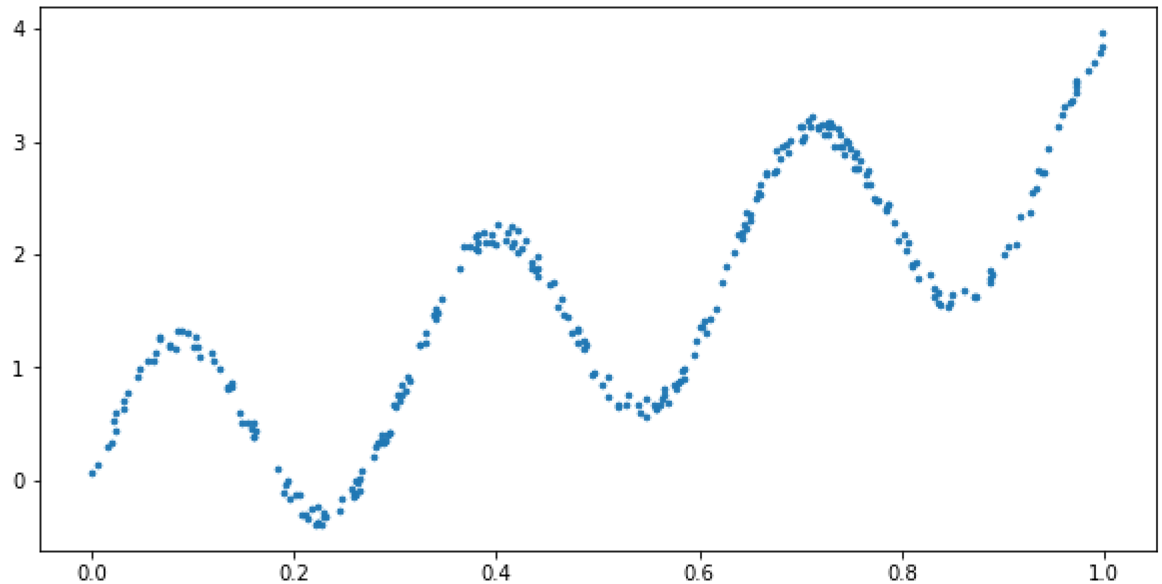
In [1]:
```
import numpy as np
from math import *
import matplotlib.pyplot as plt
```

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]:
```
0.004999281432652347 5
```

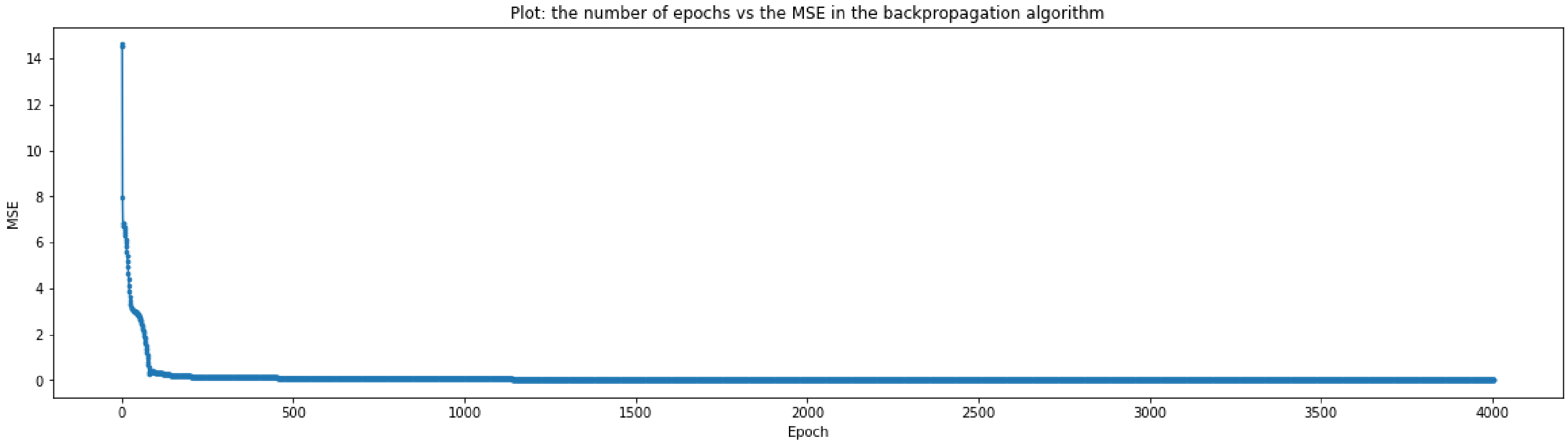
Q 4

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()
```