A2

In [1]:
```
# Standard imports
import numpy as np
import matplotlib.pylab as plt
%matplotlib inline
import importlib
import time
```

Uncomment these to use the solution instead of your own implementation

In [2]:
```
from a2_solutions import FeedForward
from a2_solutions import BackProp
from a2_solutions import Learn
```

Q1: Logistic Function

$$\sigma(z) = \frac{1}{1 - e^{-z}}$$

$$\frac{\partial(1 + e^{-z})}{\partial z} = -(e^{-z}) = -e^{-z}$$

$$\frac{\partial\sigma(z)}{\partial z} = \frac{\frac{\partial 1}{\partial z} \cdot (1 + e^{-z}) - \frac{\partial(1+e^{-z})}{\partial z} \cdot 1}{\sigma(z)^2} = \frac{e^{-z}}{(1 + e^{-z})^2}$$

$$= \frac{1}{(1 + e^{-z})} \frac{(1 + e^{-z}) - 1}{(1 + e^{-z})} = \frac{1}{(1 + e^{-z})} \left(1 - \frac{(1}{(1 + e^{-z})}\right) = \sigma(z)(1 - \sigma(z))$$

To help you with $\LaTeX$, and to show you my expectations, here is a sample taken from the lecture notes, taken from the 3rd and 4th page of the notes entitled "Error Backpropagation". It has nothing to do with the solution to this question, but just demonstrates some of the features of $\LaTeX$. Notice how I include English statments to guide the reader through the derivation.

[This web page (http://detexify.kirelabs.org/classify.html)](http://detexify.kirelabs.org/classify.html) is very handy for identifying $\LaTeX$ symbols.

More generally, for $\vec{x} \in \mathbb{R}^X$, $\vec{h} \in \mathbb{R}^H$, and $\vec{y} \in \mathbb{R}^Y$.

$$
\begin{aligned}
\frac{\partial E}{\partial \alpha_i} &= \frac{dh_i}{d\alpha_i} \\
&= \frac{dh_i}{d\alpha_i}[M_{1i} \ \cdots \ M_{Yi}] \cdot \left[\frac{\partial E}{\partial \beta_1} \ \cdots \ \frac{\partial E}{\partial \beta_Y}\right] \\
&= \frac{dh_i}{d\alpha_i}[M_{1i} \ \cdots \ M_{Yi}] \begin{bmatrix} \frac{\partial E}{\partial \beta_1} \\ \vdots \\ \frac{\partial E}{\partial \beta_Y} \end{bmatrix}
\end{aligned}
$$

Thus, for all elements,

$$
\begin{bmatrix} \frac{\partial E}{\partial \alpha_1} \\ \vdots \\ \frac{\partial E}{\partial \alpha_H} \end{bmatrix} = \begin{bmatrix} \frac{dh_1}{d\alpha_1} \\ \vdots \\ \frac{dh_H}{d\alpha_H} \end{bmatrix} \odot \begin{bmatrix} M_{11} & \cdots & M_{Y1} \\ \vdots & \ddots & \vdots \\ M_{1H} & \cdots & M_{YH} \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial \beta_1} \\ \vdots \\ \frac{\partial E}{\partial \beta_Y} \end{bmatrix}
$$

$$
\frac{\partial E}{\partial \vec{\alpha}} = \frac{d\vec{h}}{d\vec{\alpha}} \odot M^{\mathrm{T}} \frac{\partial E}{\partial \vec{\beta}}
$$

Q2: Softmax

$$
E\left(\vec{y}, \vec{t}\right) = -\sum_{k=1}^{K} t_k \ln y_k
$$

$$
y_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}
$$

$$
\frac{\partial \sum_{j=1}^{K} e^{z_j}}{\partial z_j} = \frac{\partial \sum_{k=1}^{K} e^{z_k}}{e^{z_j}} \frac{e^{z_j}}{z_j} = 1 \cdot e^{z_j} = e^{z_j}
$$

$$
\frac{\partial y_k}{\partial z_j} = \frac{\frac{\partial e^{z_k}}{\partial z_j}\left(\sum_{j=1}^{K} e^{z_j}\right) - (e^{z_k})\frac{\partial \sum_{j=1}^{K} e^{z_j}}{\partial z_j}}{\left(\sum_{j=1}^{K} e^{z_j}\right)^2} = \begin{cases} \frac{0 \cdot \left(\sum_{j=1}^{K} e^{z_j} - e^{z_k}\right) - e^{z_k} e^{z_j}}{\left(\sum_{j=1}^{K} e^{z_j}\right)^2} = \frac{-e^{z_k} e^{z_j}}{\left(\sum_{j=1}^{K} e^{z_j}\right)^2} = -y_j y_k \\ \frac{e^{z_j} \sum_{k=1}^{j} e^{z_j} - e^{z_j}}{\left(\sum_{j=1}^{K} e^{z_j}\right)^2} = \frac{e^{2z_j}}{\left(\sum_{j=1}^{K} e^{z_j}\right)^2} - \frac{e^{z_j}}{\left(\sum_{j=1}^{K} e^{z_j}\right)^2} = y_j(1 - \end{cases}
$$

$$
\frac{\partial E}{\partial z_j} = -\sum_{k=1}^{K} t_k \frac{\partial \ln y_k}{\partial z_j} = -\sum_{k=1}^{K} t_k \frac{\partial \ln y_k}{\partial y_k}\frac{\partial y_k}{\partial z_j} = -\sum_{k=1}^{K} \frac{t_k}{y_k}\frac{\partial y_k}{\partial z_j} = -\left[\left(\sum_{k=1}^{K} \frac{t_k}{y_k}\left(-y_j y_k\right)\right) + \frac{t_j}{y_j}
$$

Q3: Top-Layer Error Gradients

a

$$\frac{\partial E(y,t)}{\partial y} = \frac{\partial}{\partial y}(-t\ln y - (1-t)\ln(1-y)) = \frac{-t}{y} - \frac{1-t}{1-y}(-1) = \frac{y-t}{y(1-y)}$$
$$\frac{\partial y}{\partial z} = \sigma(z)\,(1-\sigma(z))$$
$$\frac{\partial E}{\partial z} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial z} = \frac{\sigma(z)-t}{\sigma(z)(1-\sigma(z))}[\sigma(z)\,(1-\sigma(z))] = \sigma(z)-t$$

b

$$\frac{\partial E(y,t)}{\partial y} = \frac{\partial}{\partial y}\frac{1}{2}(y-t)^2 = 2(y-t)$$
$$y = \sigma(z) = z$$
$$\frac{\partial y}{\partial z} = 1$$
$$\frac{\partial E}{\partial z} = \frac{1}{2}\frac{\partial E}{\partial y}\frac{\partial y}{\partial z} = (y-t) = (z-t)$$

Q4: Implementing Backprop

Supplied Helper Functions

In [3]:
```python
# Supplied functions

def NSamples(x):
    '''
        n = NSamples(x)

        Returns the number of samples in a batch of inputs.

        Input:
         x   is a 2D array

        Output:
         n   is an integer
    '''
    return len(x)

def OneHot(z):
    '''
        y = OneHot(z)

        Applies the one-hot function to the vectors in z.
        Example:
          OneHot([[0.9, 0.1], [-0.5, 0.1]])
          returns np.array([[1,0],[0,1]])

        Input:
         z    is a 2D array of samples

        Output:
         y    is an array the same shape as z
    '''
    y = []
    # Locate the max of each row
    for zz in z:
        idx = np.argmax(zz)
        b = np.zeros_like(zz)
        b[idx] = 1.
        y.append(b)
    y = np.array(y)
    return y
```

4(a)

In [4]:
```
# Grading:
# [1] Divide each by NSamples(t) or NSamples(y) to get the mean
# Plus one mark for each of the 4 formulas, as indicated below.
def CrossEntropy(y, t):
    '''
        E = CrossEntropy(y, t)

        Evaluates the mean cross entropy loss between outputs y and targets

        Inputs:
          y is an array holding the network outputs
          t is an array holding the corresponding targets

        Outputs:
          E is the mean CE
    '''

    # === YOUR CODE HERE ===

    return - np.sum(t * np.log(y) + (1-t)* np.log(1-y))/ NSamples(y)


def gradCrossEntropy(y, t):
    '''
        E = gradCrossEntropy(y, t)

        Given targets t, evaluates the gradient of the mean cross entropy l
        with respect to the output y.

        Inputs:
          y is the array holding the network's output
          t is an array holding the corresponding targets

        Outputs:
          dEdy is the gradient of CE with respect to output y
    '''

    # === YOUR CODE HERE ===
    return ((y-t)/(y*(1-y))) / NSamples(y)


def MSE(y, t):
    '''
        E = MSE(y, t)

        Evaluates the mean squared error loss between outputs y and targets

        Inputs:
          y is the array holding the network's output
          t is an array holding the corresponding targets

        Outputs:
          E is the MSE
    '''

    # === YOUR CODE HERE ===
```

```
    return 1 / 2 * np.sum((y - t)**2) / NSamples(y)


def gradMSE(y, t):
    '''
        E = gradMSE(y, t)

        Given targets t, evaluates the gradient of the mean squared error l
        with respect to the output y.

        Inputs:
          y is the array holding the network's output
          t is an array holding the corresponding targets

        Outputs:
          dEdy is the gradient of MSE with respect to output y
    '''

    # === YOUR CODE HERE ===
    return (y - t)/ NSamples(y)
```

In [5]:

```
#=================================================================
#
#  UNCOMMENT THE CORRESPONDING LINES BELOW IF YOU WANT TO USE
#  THE SOLUTIONS INSTEAD OF YOUR VERSION.
#
#=================================================================
from a2_solutions import CrossEntropy
from a2_solutions import gradCrossEntropy
from a2_solutions import MSE
from a2_solutions import gradMSE
```

In [ ]:

Layer Class

In [6]:
```
class Layer():

    def __init__(self, n_nodes, act='logistic'):
        '''
            lyr = Layer(n_nodes, act='logistic')

            Creates a layer object.

            Inputs:
             n_nodes  the number of nodes in the layer
             act      specifies the activation function
                      Use 'logistic' or 'identity'
        '''
        self.N = n_nodes  # number of nodes in this layer
        self.h = []       # node activities
        self.z = []
        self.b = np.zeros(self.N)  # biases

        # Activation functions
        self.sigma = self.Logistic
        self.sigma_p = (lambda : self.Logistic_p())
        if act == 'identity':
            self.sigma = self.Identity
            self.sigma_p = (lambda : self.Identity_p())

    def Logistic(self):
        return 1. / (1. + np.exp(-self.z))
    def Logistic_p(self):
        return self.h * (1.-self.h)
    def Identity(self):
        return self.z
    def Identity_p(self):
        return np.ones_like(self.h)
```

4(b,c,d) Network Class

In [7]:
```
class Network():


    def FeedForward(self, x):
        '''
            y = net.FeedForward(x)

            Runs the network forward, starting with x as input.
            Returns the activity of the output layer.

            All node use
            Note: The activation function used for the output layer
            depends on what self.Loss is set to.
        '''
        try: FeedForward
        except NameError:

            #========= YOUR IMPLEMENTATION BELOW =========

            x = np.array(x)  # Convert input to array, in case it's not

            # === YOUR CODE HERE ===
            self.lyr[0].h = x
            for i in range(0,self.n_layers-1):
                # next layer: z = wx + b, y = sigma(z)
                # w,x,b from this layer.
                self.lyr[i+1].z = self.lyr[i].h.dot(self.W[i]) + self.lyr[i
                self.lyr[i+1].h = self.lyr[i+1].sigma()
            return self.lyr[-1].h

            #========= YOUR IMPLEMENTATION ABOVE =========

        else:
            return FeedForward(self, x)


    def BackProp(self, t, lrate=0.05):
        '''
            net.BackProp(targets, lrate=0.05)

            Given the current network state and targets t, updates the conn
            weights and biases using the backpropagation algorithm.

            Inputs:
             t      an array of targets (number of samples must match the
                    network's output)
             lrate  learning rate
        '''
        #====== REMOVE BELOW IF YOU DON'T PLAN TO USE THE SOLUTIONS ======
        try: BackProp
        except NameError:

            #========= YOUR IMPLEMENTATION BELOW =========

            t = np.array(t)  # convert t to an array, in case it's not
```

```
        # === YOUR CODE HERE ===
        n = NSamples(self.lyr[-1].h)
        dE_dz = (self.lyr[-1].h - t)/n
        for i in range(self.n_layers - 2, -1, -1):
            dE_dw = (self.lyr[i].h.T).dot(dE_dz)
            dE_dz = self.lyr[i].sigma_p() * dE_dz.dot(self.W[i].T)
            # update
            self.W[i] = self.W[i] - lrate * dE_dw



        #========= YOUR IMPLEMENTATION ABOVE =========

    else:
        BackProp(self, t, lrate)


def Learn(self, inputs, targets, lrate=0.05, epochs=1, progress=True):
    '''
        Network.Learn(data, lrate=0.05, epochs=1, progress=True)

        Run through the dataset 'epochs' number of times, incrementing
        network weights after each epoch. For each epoch, it
        shuffles the order of the samples.

        Inputs:
          data is a list of 2 arrays, one for inputs, and one for targe
          lrate is the learning rate (try 0.001 to 0.5)
          epochs is the number of times to go through the training data
          progress (Boolean) indicates whether to show cost
    '''
    try: Learn
    except NameError:

        #========= YOUR IMPLEMENTATION BELOW =========

        # === YOUR CODE HERE ===

        epoch_idx = 0
        while epoch_idx < epochs:
            self.cost_history.append(MSE(self.FeedForward(inputs),targe
            self.BackProp(targets, lrate)
            epoch_idx += 1




        #========= YOUR IMPLEMENTATION ABOVE =========

    else:
        Learn(self, inputs, targets, lrate=lrate, epochs=epochs, progre


def __init__(self, sizes, type='classifier'):
    '''
        net = Network(sizes, type='classifier')
```

```
            Creates a Network and saves it in the variable 'net'.

            Inputs:
              sizes is a list of integers specifying the number
                  of nodes in each layer
                  eg. [5, 20, 3] will create a 3-layer network
                      with 5 input, 20 hidden, and 3 output nodes
              type can be either 'classifier' or 'regression', and
                  sets the activation function on the output layer,
                  as well as the loss function.
                  'classifier': logistic, cross entropy
                  'regression': linear, mean squared error
        '''
        self.n_layers = len(sizes)
        self.lyr = []    # a list of Layers
        self.W = []      # Weight matrices, indexed by the layer below it

        self.cost_history = []  # keeps track of the cost as learning progr

        # Two common types of networks
        # The member variable self.Loss refers to one of the implemented
        # loss functions: MSE, or CrossEntropy.
        # Call it using self.Loss(t)
        if type=='classifier':
            self.classifier = True
            self.Loss = CrossEntropy
            self.gradLoss = gradCrossEntropy
            activation = 'logistic'
        else:
            self.classifier = False
            self.Loss = MSE
            self.gradLoss = gradMSE
            activation = 'identity'

        # Create and add Layers (using logistic for hidden layers)
        for n in sizes[:-1]:
            self.lyr.append( Layer(n) )

        # For the top layer, we use the appropriate activtaion function
        self.lyr.append( Layer(sizes[-1], act=activation) )

        # Randomly initialize weight matrices
        for idx in range(self.n_layers-1):
            m = self.lyr[idx].N
            n = self.lyr[idx+1].N
            temp = np.random.normal(size=[m,n])/np.sqrt(m)
            self.W.append(temp)

    def Evaluate(self, inputs, targets):
        '''
            E = net.Evaluate(data)

            Computes the average loss over the supplied dataset.

            Inputs
             inputs  is an array of inputs
             targets is a list of corresponding targets
```

```
            Outputs
             E is a scalar, the average loss
        '''
        y = self.FeedForward(inputs)
        return self.Loss(y, targets)

    def ClassificationAccuracy(self, inputs, targets):
        '''
            a = net.ClassificationAccuracy(data)

            Returns the fraction (between 0 and 1) of correct one-hot class
            in the dataset.
        '''
        y = self.FeedForward(inputs)
        yb = OneHot(y)
        n_incorrect = np.sum(yb!=targets) / 2.
        return 1. - float(n_incorrect) / NSamples(inputs)
```

Classification

Create a Classification Dataset

In [8]:
```
# 5 Classes in 8-Dimensional Space
np.random.seed(15)
noise = 0.1
InputClasses = np.array([[1,0,1,0,0,1,1,0],
                         [0,1,0,1,0,1,0,1],
                         [0,1,1,0,1,0,0,1],
                         [1,0,0,0,1,0,1,1],
                         [1,0,0,1,0,1,0,1]], dtype=float)
OutputClasses = np.array([[1,0,0,0,0],
                          [0,1,0,0,0],
                          [0,0,1,0,0],
                          [0,0,0,1,0],
                          [0,0,0,0,1]], dtype=float)
n_input = np.shape(InputClasses)[1]
n_output = np.shape(OutputClasses)[1]
n_classes = np.shape(InputClasses)[0]

# Create a training dataset
n_samples = 100
training_output = []
training_input = []
for idx in range(n_samples):
    k = np.random.randint(n_classes)
    x = InputClasses[k,:] + np.random.normal(size=n_input)*noise
    t = OutputClasses[k,:]
    training_input.append(x)
    training_output.append(t)

# Create a test dataset
n_samples = 100
test_output = []
test_input = []
for idx in range(n_samples):
    k = np.random.randint(n_classes)
    x = InputClasses[k,:] + np.random.normal(size=n_input)*noise
    t = OutputClasses[k,:]
    test_input.append(x)
    test_output.append(t)

train = [np.array(training_input), np.array(training_output)]
test = [np.array(test_input), np.array(test_output)]
```

Neural Network Model

In [9]:
```
# Create a Network
net = Network([n_input, 18, n_output], type='classifier')
```

In [10]:
```
CE = net.Evaluate(train[0], train[1])
```

```
a2_solutions.FeedForward
a2_solutions.CrossEntropy
```

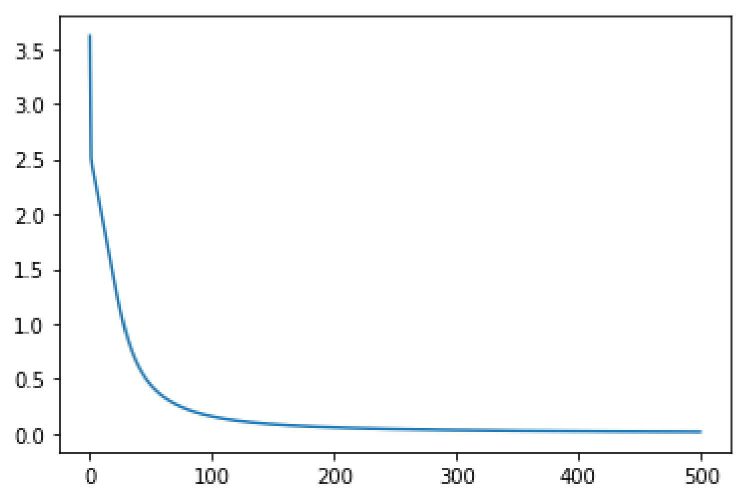
In [11]:
```
# Evaluate it before training
CE = net.Evaluate(train[0], train[1])
accuracy = net.ClassificationAccuracy(train[0], train[1])
print('Cross Entropy = '+str(CE))
print('     Accuracy = '+str(accuracy*100.)+'%')
```

```
a2_solutions.FeedForward
a2_solutions.CrossEntropy
a2_solutions.FeedForward
Cross Entropy = 3.6170513253334455
     Accuracy = 26.0%
```

In [12]:
```
net.Learn(train[0], train[1], epochs=500, lrate=1.)
```

```
a2_solutions.Learn
a2_solutions.FeedForward
a2_solutions.BackProp
a2_solutions.gradCrossEntropy
a2_solutions.CrossEntropy
Epoch 0: Cost = 3.6170513253334455
a2_solutions.FeedForward
a2_solutions.BackProp
a2_solutions.gradCrossEntropy
a2_solutions.CrossEntropy
a2_solutions.FeedForward
a2_solutions.BackProp
a2_solutions.gradCrossEntropy
a2_solutions.CrossEntropy
a2_solutions.FeedForward
a2_solutions.BackProp
a2_solutions.gradCrossEntropy
a2_solutions.CrossEntropy
a2_solutions.FeedForward
```

In [13]:
```
plt.plot(net.cost_history);
```



Evaluate it After Training

In [14]:
```
print('Training Set')
CE = net.Evaluate(train[0], train[1])
accuracy = net.ClassificationAccuracy(train[0], train[1])
print('Cross Entropy = '+str(CE))
print('     Accuracy = '+str(accuracy*100.)+'%')
```

```
Training Set
a2_solutions.FeedForward
a2_solutions.CrossEntropy
a2_solutions.FeedForward
Cross Entropy = 0.01766166130802752
     Accuracy = 100.0%
```

In [15]:
```
print('Test Set')
CE = net.Evaluate(test[0], test[1])
accuracy = net.ClassificationAccuracy(test[0], test[1])
print('Cross Entropy = '+str(CE))
print('     Accuracy = '+str(accuracy*100.)+'%')
```

```
Test Set
a2_solutions.FeedForward
a2_solutions.CrossEntropy
a2_solutions.FeedForward
Cross Entropy = 0.018996107802952165
     Accuracy = 100.0%
```

In [16]:
```
p = np.random.randint(len(test[0]))
print(net.FeedForward(test[0][p]))
print(test[1][p])
```

```
a2_solutions.FeedForward
[6.69824684e-05 9.83005841e-01 1.28793238e-02 2.22796938e-04
 2.45456803e-03]
[0. 1. 0. 0. 0.]
```

Regression

Create a Regression Dataset

In [17]:
```
# 1D -> 1D (linear mapping)
np.random.seed(846)
n_input = 1
n_output = 1
slope = np.random.rand() - 0.5
intercept = np.random.rand()*2. - 1.

def myfunc(x):
    return slope*x+intercept

# Create a training dataset
n_samples = 200
training_output = []
training_input = []
xv = np.linspace(-1, 1, n_samples)
for idx in range(n_samples):
    #x = np.random.rand()*2. - 1.
    x = xv[idx]
    t = myfunc(x) + np.random.normal(scale=0.1)
    training_input.append(np.array([x]))
    training_output.append(np.array([t]))

# Create a testing dataset
n_samples = 50
test_input = []
test_output = []
xv = np.linspace(-1, 1, n_samples)
for idx in range(n_samples):
    #x = np.random.rand()*2. - 1.
    x = xv[idx] + np.random.normal(scale=0.1)
    t = myfunc(x) + np.random.normal(scale=0.1)
    test_input.append(np.array([x]))
    test_output.append(np.array([t]))

# Create a perfect dataset
n_samples = 100
perfect_input = []
perfect_output = []
xv = np.linspace(-1, 1, n_samples)
for idx in range(n_samples):
    #x = np.random.rand()*2. - 1.
    x = xv[idx]
    t = myfunc(x)
    perfect_input.append(np.array([x]))
    perfect_output.append(np.array([t]))

train = [np.array(training_input), np.array(training_output)]
test = [np.array(test_input), np.array(test_output)]
perfect = [np.array(perfect_input), np.array(perfect_output)]
```

Neural Network Model

In [18]:
```
net = Network([1, 10, 1], type='regression')
```

In [19]:
```
# Evaluate it before training
mse = net.Evaluate(train[0], train[1])
print('MSE = '+str(mse))
```

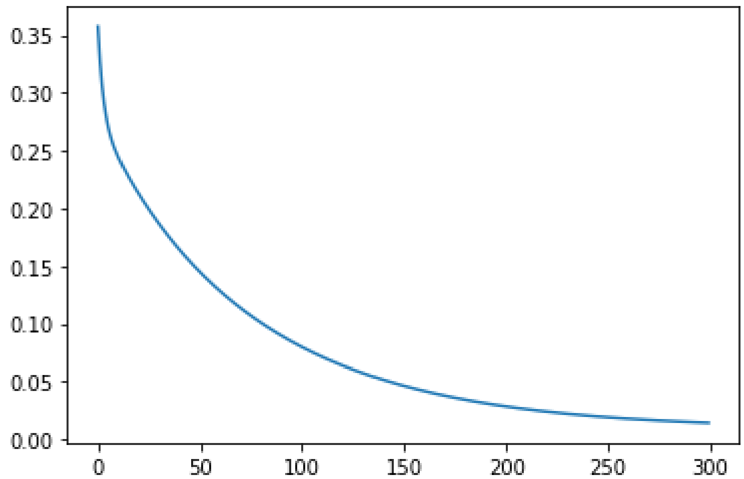
```
a2_solutions.FeedForward
a2_solutions.MSE
MSE = 0.35753196157541495
```

Training

In [20]:
```
net.Learn(train[0], train[1], epochs=300)
```

...

In [21]:
```
plt.plot(net.cost_history);
```



Evaluate it After Training

In [22]:
```
# On training dataset
mse = net.Evaluate(train[0], train[1])
print('Training MSE = '+str(mse))
```

```
a2_solutions.FeedForward
a2_solutions.MSE
Training MSE = 0.014130050117871617
```

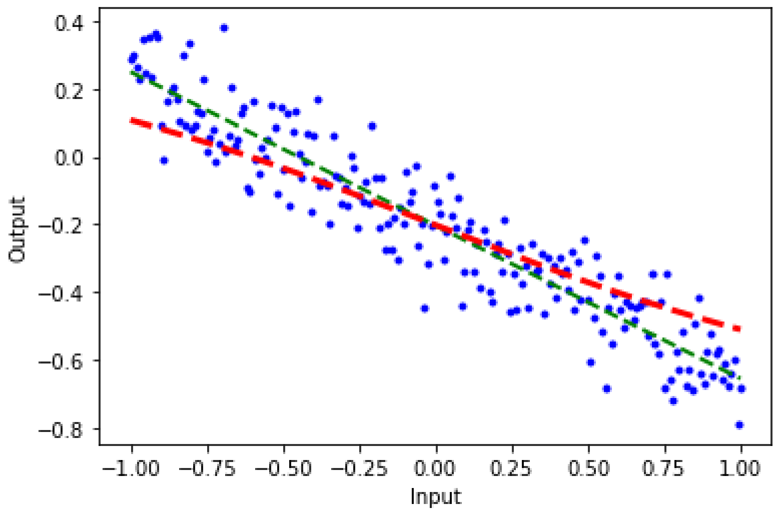
In [23]:
```
# On test dataset
mse = net.Evaluate(test[0], test[1])
print('Test MSE = '+str(mse))
```

```
a2_solutions.FeedForward
a2_solutions.MSE
Test MSE = 0.017301624623795225
```

In [24]:
```
# Evaluate our model and the TRUE solution (since we know it)
s = np.linspace(-1, 1, 200)
y = net.FeedForward(np.array([s]).T)
p = [myfunc(x) for x in s]
```

```
a2_solutions.FeedForward
```

In [25]:
```
# Plot the training data,
# as well as out model and the true model
plt.plot(training_input, training_output, 'b.')
plt.plot(s,p, 'g--', linewidth=2)
plt.plot(s,y, 'r--', linewidth=3)
plt.xlabel('Input')
plt.ylabel('Output');
```



In [ ]:

In [ ]:

In [ ]: