PROJECT 1

COSC 525 - DEEP LEARNING

AUTHORS

KEVIN DE ANGELI

The University of Tennessee
Knoxville

JANUARY 28, 2020

1 Introduction

This project explores some the basis of the perceptron (only one neuron) and feedforward neural networks (network of arbitrary size). The program (see Appendix) was written using three different classes: Neuron, FullyConnectedLayer, and NeuralNetwork. This is not the most efficient way to program a neural network. However, by implementing the model in this fashion, I was required to fully understand how different elements of the network work together. More precisely, I connected individual ideas such as loss and activation function, their derivatives, gradient descent and backpropagation. A lot of most efficient models can be found online, but coming up with an innovative design really helps you dive into deep learning. Overall, the program computes the output that are required for this project (I have not tested the program on more complex problems).

2 Assumptions/Choices made

I realized that instead of passing the "input size" and the number of layers individually, one can pass an array where the first and last number represent the input size and the output, respectively, and the number in the middle represent the number of neuron in each of the hidden layer; i.e. [2,3,3,2] represents a network with an input size of 2, and output size of 3 and two hidden layers with 3 neurons in each.

I have also decided to define "by default" parameters for the network, so that if you just call the network without specific values, the program would automatically assign them. This was particularly for the activation function (Sigmoid by default) and the loss function (MSE by default).

3 Problems/Issues not resolved

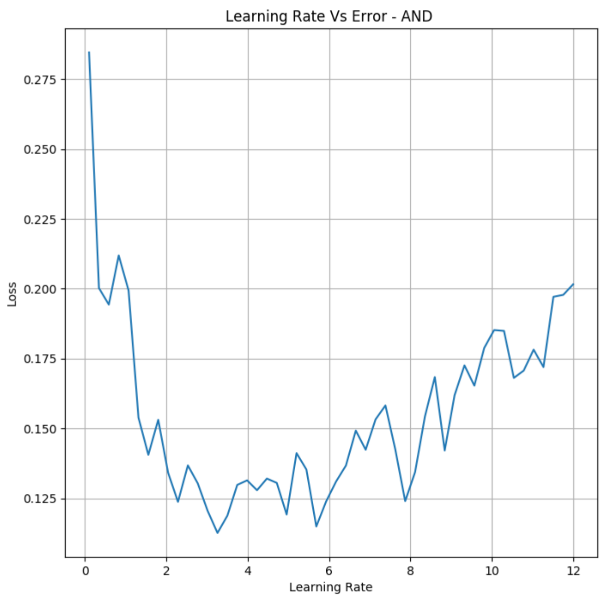
4 Running the Code: Instructions

From the console you can just type "python3 main.py TASK". Where TASK can be "example, "xor", "and", "lossLearning", or "lossEpoch". The first three are part of the requirements for the project, and "lossLearning" and "lossEpoch" executes the functions that plots the error graphs shown in the next section.

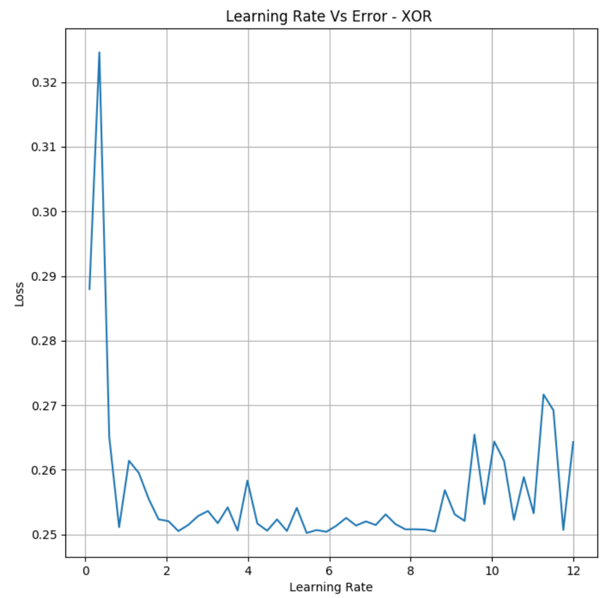
5 Loss plots

For each plot, 50 different models were defined with different learning rates between 0.1 and 12. For each learning rate, the model was fed the data 10 times, then them model was tested in the 4 data points. The average loss of these four data points was averaged and stored in a list. The final vector contained the average of the 50 learning rates. As one would expect, the loss is high when the learning rate is close to 0, since it's learning really slowly and 10 iterations is not enough to approximate the parameters that would reduce the error. For the AND plot, the optimal learning rate seems to be close to 5.8, and for the XOR plot, there seems to be a couple of optimal learning rates: 2, 2.9, and even 4. I think that since the XOR problem is non-linear, one would expect much more noise with respect to the learning parameter and the loss. Figure 1b shows much higher variability.

I have also decided to include a plot of Learning Rate vs Number of Epochs (Figure 2). This plot shows that it takes around 10 to 20 epochs for the MSE to stabilize. This plot was run with what appears to be the optimal learning rate (according to Figure 1).

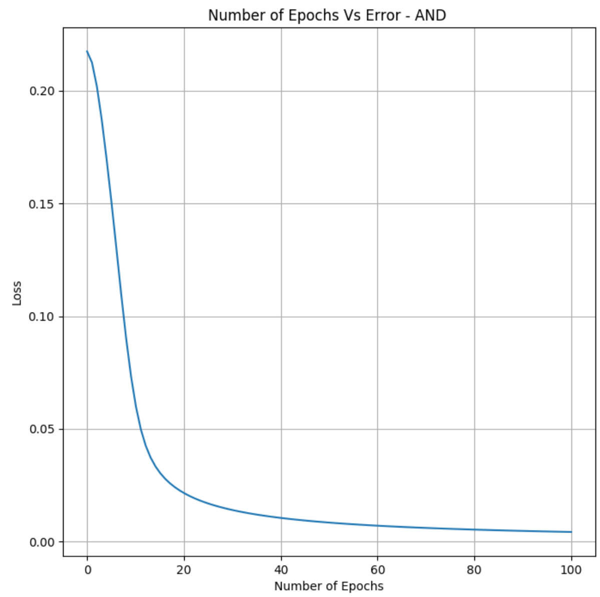


(a) AND

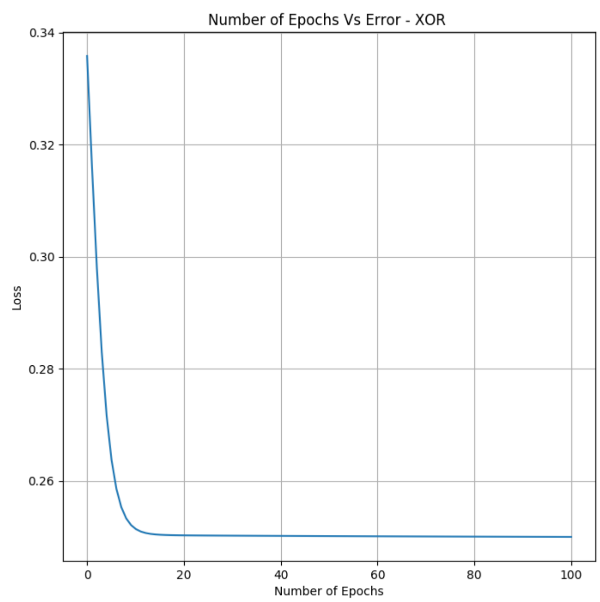


(b) XOR

Figure 1: Learning Rate vs Loss. MSE was used to compute loss, and Sigmoid was used as the activation function.



(a) AND. Learning Rate = 5.5



(b) XOR. Learning Rate = 2

Figure 2: Epoch vs Loss. MSE was used to compute loss, and Sigmoid was used as the activation function.

6 Appendix

6.1 Python Script

6.1.1 main.py

```python
'''
Created by: Kevin De Angeli
Email: kevindeangeli@utk.edu
Date: 2020-01-15

Note: The activation and loss function are in two separate files
which will be included in the zip file.

General description:
The NeuralNetwork objects will contain objects of the FullyConnectedLayer class.
The FullyConnectedLayer contains objects of the Neuron class. Each of the Neuron
objects contains their own weights, biase, and delta value.

I created a method in the Neuron function called "mini_Delta" which is used to
compute the weight (given a certain index) times the delta of that neuron. This
is used for backpropagation. The backpropagation function in the NeuralNetwork first
computes the Sum Delta values for the layers, and these are passed to individual
layers so that the weights of each neuron can be updated.

'''


from errorFunctions import * #I put the error functions and their derivatives in a different file.
from activation import *     #I put the activation functions and their derivatives in a different file.

import numpy as np
import sys
import matplotlib.pyplot as plt

class Neuron():

    def __init__(self,inputLen, activationFun = "sigmoid", lossFunction="mse" , learningRate = .5, weights = None, bias = None):
        self.inputLen = inputLen
        self.learnR = learningRate
        self.activationFunction = activationFun
        self.lossFunction = lossFunction

```

```python
        self.output = None #Saving the output of the neuron (after feedforward) makes things easy for backprop
        self.input = None   #Saves the input to the neuron for backprop.
        self.newWeights = [] #Saves new weight but it doesn't update until the end of backprop. (method: updateWeight)
        self.newBias = None
        self.delta = None #individual deltas required for backprop.

        if weights is None:
            #set them to random:
            self.weights = [np.random.random_sample() for i in range(inputLen)]
            self.bias = np.random.random_sample()

        else:
            self.weights = weights
            self.bias = bias

        #this series of if statement define the activation and loss functions, and their derivatives.
        if activationFun is "sigmoid":
            self.activate = sigmoid
            self.activation_prime = sigmoid_prime
        else:
            self.activate = linear

        if lossFunction is "mse":
            self.loss = mse
            self.loss_prime= mse_prime
        else:
            self.loss = crossEntropy
            self.loss_prime = crossEntropy_prime

    #The following pictures will be defined based on the parameters
    #that is passed to the object.
    def activate(self):
        pass
    def loss(self):
        pass
    def activation_prime(self):
        pass
    def loss_prime(self):
        pass

    #This function is called after backpropagation.
    def updateWeight(self):
        self.weights = self.newWeights
```

```
        self.newWeights = []
        self.bias = self.newBias
        self.newBias = None

    def calculate(self, input):
        '''
        Given an input, it will calculate the output
        :return:
        '''
        self.input = input
        self.output = self.activate(np.dot(input,self.weights) + self.bias)
        return self.output

    #The delta of the last layer is computed a little different, so it has its own function.
    def backpropagationLastLayer(self, target):
        self.delta = self.loss_prime(self.output, target) * self.activation_prime(self.output)
        x1=  self.loss_prime(self.output, target)
        x2= self.activation_prime(self.output)
        self.newBias = self.bias - self.learnR*self.delta
        for index, PreviousNeuronOutput in enumerate(self.input):
            self.newWeights.append(self.weights[index] - self.learnR * self.delta * PreviousNeuronOutput)

    def backpropagation(self, sumDelta):
        #sumDelta will be computed at the layer level. Since it requires weights from multiple neurons.
        self.delta = sumDelta * self.activation_prime(self.output)
        self.newBias = self.bias - self.learnR * self.delta
        for index, PreviousNeuronOutput in enumerate(self.input):
            self.newWeights.append(self.weights[index] - self.learnR * self.delta * self.input[index])

    #Used to compute the sumation of the Deltas for backprop.
    def mini_Delta(self, index):
        return self.delta * self.weights[index]



class FullyConnectedLayer():
    def __init__(self, inputLen, numOfNeurons = 5, activationFun = "sigmoid", lossFunction= "mse", learningRate = .1, weights = None, bias = None):
        self.inputLen = inputLen
        self.neuronsNum = numOfNeurons
        self.activationFun = activationFun
```

```python
        self.learningRate = learningRate
        self.weights = weights
        self.bias = bias
        self.layerOutput = []
        self.lossFunction = lossFunction

        #Random weights or user defined weights:
        if weights is None:
            self.neurons = [Neuron(inputLen=self.inputLen, activationFun=activationFun,lossFunction=self.lossFunction ,learningRate=self.learningRate, weights=self.weights) for i in range(numOfNeurons)]
        else:
            self.neurons = [Neuron(inputLen=self.inputLen, activationFun=activationFun,lossFunction=self.lossFunction, learningRate=self.learningRate, weights=self.weights[i], bias=self.bias[i]) for i in range(numOfNeurons)]


    def calculate(self, input):
        '''
        Will calculate the output of all the neurons in the layer.
        :return:
        '''
        self.layerOutput = []
        for neuron in self.neurons:
            self.layerOutput.append(neuron.calculate(input))

        return self.layerOutput

    def backPropagateLast(self, target):
        for targetIndex, neuron in enumerate(self.neurons):
            neuron.backpropagationLastLayer(target=target[targetIndex])

    def updateWeights(self):
        for neuron in self.neurons:
            neuron.updateWeight()

    #Computes the sum of the deltas times their weights based on the number of neurons in the previous layer.
    def deltaSum(self):
        delta_sumArr  = []
        x=len(self.neurons[0].weights)
        for i in range(len(self.neurons[0].weights)): #Number of Weights in the RightLayer = Number of neurons in the LeftLayer
            delta_sum = 0
```

```
            for index, neuron in enumerate(self.neurons):
                delta_sum += neuron.mini_Delta(i)
            delta_sumArr.append(delta_sum)
        return delta_sumArr

    def backpropagation(self, deltaArr):
        #Each neuron needs a delta to update their weights:
        for index, neuron in enumerate(self.neurons):
            neuron.backpropagation(deltaArr[index])


class NeuralNetwork():
    def __init__(self, neuronsNum = None, activationVector = 0, lossFunction = "mse", learningRate = .1, weights = None, bias = None):
        self.inputLen   = neuronsNum[0]
        self.layersNum  = len(neuronsNum)-1 #Don't count the first one (input).
        self.activationVector = activationVector
        self.lossFunction = lossFunction
        self.learningRate = learningRate
        self.weights = weights
        self.bias = bias

        if neuronsNum is None :  #By default, each layer will have 3 neurons, unless specified.
            self.neuronsNum = [3 for i in range(layersNum)]
        else:
            self.neuronsNum = neuronsNum[1:len(neuronsNum)] #Don't count the first one (input)

        if activationVector is None or activationVector != self.layersNum: #This is the default vector if one is not provided when the class is created.
            self.activationVector = ["sigmoid" for i in range(self.layersNum)]

        #Define the layers of the networks with the respective neurons:
        self.layers = []
        inputLenLayer = self.inputLen
        #This convoluted loop creates the layers and neurons with the appropite number of weights in each.
        if weights is None:
            for i in range(self.layersNum):
                self.layers.append(
                    FullyConnectedLayer(numOfNeurons=self.neuronsNum[i], activationFun=self.activationVector[i],lossFunction=self.lossFunctio
                    inputLen=inputLenLayer, learningRate=self.learningRate, weights=self.weights))
```

```
                # The number of weights in one layer depends on the number of neurons in the previous layer:
                inputLenLayer = self.neuronsNum[i]

        #Used defined weights:
        else:
            for i in range(self.layersNum):
                self.layers.append(
                    FullyConnectedLayer(numOfNeurons=self.neuronsNum[i], activationFun=self.activationVector[i], inputLen=inputLenLayer, learningRate=self.learningRate, weights=self.weights[i], bias=self.bias[i]))
                # The number of weights in one layer depends on the number of neurons in the previous layer:
                inputLenLayer = self.neuronsNum[i]



    def showWeights(self):
        #Function which just goes through each neuron in each layer and displays the weights.
        inputLenLayer = self.inputLen
        for i in range(self.layersNum):
            print(" ")
            for k in range(self.neuronsNum[i]):
                print(self.layers[i].neurons[k].weights)

            inputLenLayer = self.neuronsNum[i]

    def showBias(self):
        #Function which just goes through each neuron in each layer and displays the bias.
        inputLenLayer = self.inputLen
        for i in range(self.layersNum):
            #print(" ")
            for k in range(self.neuronsNum[i]):
                print(self.layers[i].neurons[k].bias)

            inputLenLayer = self.neuronsNum[i]

    def calculate(self, input):
        '''
        given an input calculates the output of the network.
        input should be a list.
        :return:
        '''
```

```python
        output = input
        for layer in self.layers:
            output = layer.calculate(output)

        return output

    def backPropagate(self, target):
        self.layers[-1].backPropagateLast(target)
        layersCounter = self.layersNum+1

        for i in range(2,layersCounter):
            #Calculate the sum delta for the following layer to update the previous layer.
            deltaArr = self.layers[-i + 1].deltaSum()
            self.layers[-i].backpropagation(deltaArr)

        for layer in self.layers:
            layer.updateWeights()



    def calculateLoss(self,input,target, function = "mse"):
        '''
        Given an input and desired output, calculate the loss.
        Can be implemented with MSE and binary cross.
        '''
        N = len(input)
        output = self.calculate(input)
        if function == "mse":
            error = mse(output, target)
        else:
            crossEntropy(output, target)

        return error


    def train(self, input, target, showLoss = False):
        '''
        Basically, do forward and backpropagation all together here.
        Given a single input and desired output, it will take one step of gradient descent.
        :return:
        '''
        self.calculate(input)
        if showLoss is True:
            print("mse: ", self.calculateLoss(input=input, target=target))
```

```python
        self.backPropagate(target)


def doExample():
    '''
    This function does the "Example" forward and backpop pass required for the
    assignemnt.
    '''
    print( "--- Example ---")

    #Let's try the class example by setting the bias and weights:
    Newweights = [[[.15,.20], [.25, .30]], [[.40, .45], [.5, .55]]]
    newBias = [[.35,.35],[.6,.6]]
    model = NeuralNetwork(neuronsNum=[2, 2, 2], activationVector=['sigmoid',
     'sigmoid'], lossFunction="mse",
                        learningRate=.5, weights=Newweights, bias = newBias)



    print("Original weights and biases of the network: ")
    print("Model's Weights:")
    model.showWeights()
    print("\nModel's Bias:")
    model.showBias()


    print("\nForward pass: ")
    print(model.calculate([.05,.1]))

    #model.train(input= [.05,.1], target=[.01, .99]) #you could use just this
     function to do all at once.
    model.backPropagate(target= [.01, .99])
    print("\nAfter BackProp, the updated weights are:")
    print("Model's Weights:")
    model.showWeights()
    print("\nModel's Bias:")
    model.showBias()

def doAnd():
    '''
    This function trains a single neuron to learn the "AND" logical operator.
    '''
    print( "\n--- AND ---")
    x = [[1,1],[1,0],[0,1], [0,0]]
    y = [[1],[0], [0], [0]]

```

```python
    model = NeuralNetwork(neuronsNum=[2, 1], activationVector=['sigmoid'], lossFunction="mse",
                              learningRate=6, weights=None, bias=None)


    print("-------- Before training ---------")
    print("Model's Weights:")
    model.showWeights()
    print("\nModel's Bias:")
    model.showBias()



    for i in range(10000):
        for index in range(len(x)):
            model.train(input=x[index],target=y[index])

    print("----------------------------------")
    print("\nPredictions: ")
    for index2 in range(len(x)):
        print("\nPredict: ", x[index2])
        print(model.calculate(x[index2]))

    print("-------- After training ---------")
    print("Model's Weights:")
    model.showWeights()
    print("\nModel's Bias:")
    model.showBias()




def doXor():
    '''
    This function creates two models: 1) A single neuron model which is not able to
    learn the XOR operator, and 2) A model with two neurons in the hidden layer,
    and 1 output neuron which successfully learns XOR.
    '''
    print( "\n--- XOR ---")
    x = [[1,1],[1,0],[0,1], [0,0]]
    y = [[0],[1], [1], [0]]

    print("First model: [2,1] (Single Perceptron) : \n")
    model = NeuralNetwork(neuronsNum=[2, 1], activationVector=['sigmoid'], lossFunction="mse",
                              learningRate=1.9, weights=None, bias=None)
```

```

print("-------- Before training ---------")
print("Model's Weights:")
model.showWeights()
print("\nModel's Bias:")
model.showBias()

for i in range(10000):
    for index in range(len(x)):
        model.train(input=x[index], target=y[index])

print("---------------------------------")
print("\nPredictions: ")
for index2 in range(len(x)):
    print("\nPredict: ", x[index2])
    print(model.calculate(x[index2]))

print("-------- After training ---------")
print("Model's Weights:")
model.showWeights()
print("\nModel's Bias:")
model.showBias()
print("\n\n ***************************************************\n\n")
print("Second model: [2,2,1] (Hidden layer Perceptron) : \n")

model = NeuralNetwork(neuronsNum=[2, 2, 1], activationVector=['sigmoid', 'sigmoid'], lossFunction="mse",
                      learningRate=.5, weights=None, bias=None)

print("-------- Before training ---------")
print("Model's Weights:")
model.showWeights()
print("\nModel's Bias:")
model.showBias()

for i in range(10000): #It works with 100000 and alpha = 1.5 but it takes a minute
    for index in range(len(x)):
        model.train(input=x[index],target=y[index])

print("---------------------------------")
print("\nPredictions: ")
for index2 in range(len(x)):
    print("\nPredict: ", x[index2])
    print(model.calculate(x[index2]))

```

```
    print("-------- After training ---------")
    print("Model's Weights:")
    model.showWeights()
    print("\nModel's Bias:")
    model.showBias()


def showLoss(learningRate, data = "and"):
    '''
    This function creates the Learnign Rate vs Loss plot for both: AND and XOR.
    '''
    if data is "and":
        print("Loss for AND")
        x = [[1,1],[1,0],[0,1], [0,0]]
        y = [[1],[0], [0], [0]]
        title = "LearningRateVsErrorAND_MLE.png"
        figTitle= "Learning Rate Vs Error - AND "
        modelArch = [2, 1]

    else:
        print("Loss for XOR")
        x = [[1, 1], [1, 0], [0, 1], [0, 0]]
        y = [[0], [1], [1], [0]]
        title = "LearningRateVsErrorXOR_MLE.png"
        figTitle= "Learning Rate Vs Error - XOR "
        modelArch = [2,2,1]


    errorAvrage = [] #This will contain the Ys of the plot.
    for i in learningRate:
        model  = NeuralNetwork(neuronsNum=modelArch, activationVector=['sigmoid'], lossFunction="mse",
                        learningRate=i, weights=None, bias=None)

        errorListPerLearningRate = []
        for i in range(10):
            errorList = []
            for index in range(len(x)): #Train the algorithm with entire dataset
                model.train(input=x[index], target=y[index])

            for index2 in range(len(x)):
                errorList.append(model.calculateLoss(input=x[index2], target=y[index2])) #Collect individual errors

            errorListPerLearningRate.append(np.average(errorList))

```

```
        errorAvrage.append(np.average(errorListPerLearningRate))

    fig, ax = plt.subplots(figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
    ax.plot(learningRate, errorAvrage)
    ax.set(xlabel='Learning Rate', ylabel='Loss',title=figTitle)
    ax.grid()
    plt.savefig(title)
    plt.show()

def lossVSEpoch(data="and"):
    '''
    This function creates the plots that displays the loss as a function of the
    ↪ number of epochs.
    '''
    if data is "and":
        print("Loss for AND")
        x = [[1,1],[1,0],[0,1], [0,0]]
        y = [[1],[0], [0], [0]]
        title = "EpochVsErrorAND_MLE.png"
        figTitle= "Number of Epochs Vs Error - AND "
        learnRate = 5.5
        modelArch = [2,1]

    else:
        print("Loss for XOR")
        x = [[1, 1], [1, 0], [0, 1], [0, 0]]
        y = [[0], [1], [1], [0]]
        title = "EpochVsErrorXOR_MLE.png"
        figTitle= "Number of Epochs Vs Error - XOR "
        learnRate = .5
        modelArch = [2,2,1]



    errorArr = [] #This will contain the Ys of the plot.

    model = NeuralNetwork(neuronsNum=modelArch, activationVector=None,
    ↪ lossFunction="mse",
                          learningRate=learnRate, weights=None, bias=None)
    epochsNums = 100
    for i in range(epochsNums):
        errorList = []
        for index in range(len(x)): #Train the algorithm with entire dataset
            model.train(input=x[index], target=y[index])

        for index2 in range(len(x)):
```

```
            errorList.append(model.calculateLoss(input=x[index2], target=y[index2])) #Collect individual errors

        errorArr.append(np.average(errorList))


    fig, ax = plt.subplots(figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
    ax.plot(np.linspace(0,epochsNums,epochsNums), errorArr)
    ax.set(xlabel='Number of Epochs', ylabel='Loss',title=figTitle)
    ax.grid()
    plt.savefig(title)
    plt.show()


def main():
    program_name = sys.argv[0]
    #input = sys.argv[1:] #Get input from the console.
    # Input validation:
    #if len(input) != 1:
    #    print("Input only one of these: example, and, or xor")
    #    return 0

    # This is just to run it from the editor instead of the console.
    input = ["example", "and", "xor","lossLearning", "lossEpoch"]
    input = [input[3]]

    if input[0] == "example":
        doExample()
    elif input[0] == "and":
        doAnd()
    elif input[0] == "xor":
        doXor()
    elif input[0] == "lossLearning":
        learningRateArr = np.linspace(0.1, 12, num=50)
        showLoss(learningRateArr, data="xor")
    elif input[0] == "lossEpoch":
        lossVSEpoch(data="xor")
    else:
        # Input validation
        print("Input Options: example, and, or xor")
        return 0



if __name__ == "__main__":
    main()
```

6.1.2 activation.py

```
'''
Created by: Kevin De Angeli
Email: kevindeangeli@utk.edu
Date: 2020-01-16

'''

import numpy as np


def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    x = z * (1 - z)
    return z * (1 - z)

def linear(x):
    return x
```

6.1.3 errorFunctions.py

```
'''
Created by: Kevin De Angeli
Email: kevindeangeli@utk.edu
Date: 2020-01-16

'''

import numpy as np

def crossEntropy(prediction, output):
    if output == 1:
      return -log(prediction)
    else:
      return -log(1 - prediction)

def crossEntropy_prime(prediction, output):
    return -((output/prediction) - ((1-output)/(1-prediction)))

def mse(prediction, output):
    if len(prediction) == 1:
        return (prediction[0]-output[0])**2
    else:
```

```python
        return ((prediction - output)**2).mean(axis=0)

def mse_prime(output, target):
    return output-target
```