
PROJECT 1

COSC 525 - DEEP LEARNING

AUTHORS

KEVIN DE ANGELI

*The University of Tennessee
Knoxville*

JANUARY 25, 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 back-propagation. 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

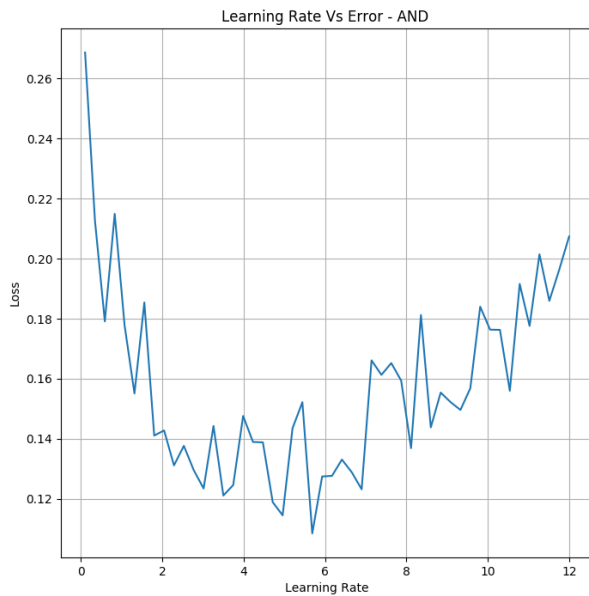
4 Running the Code: Instructions

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.

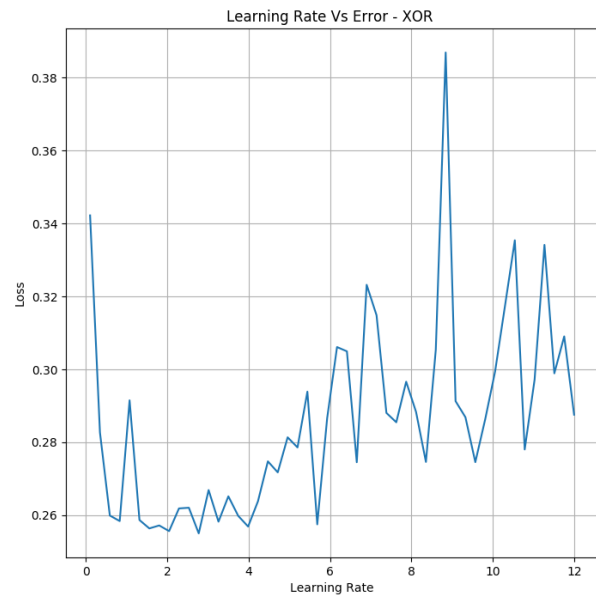
5 Loss plots

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).

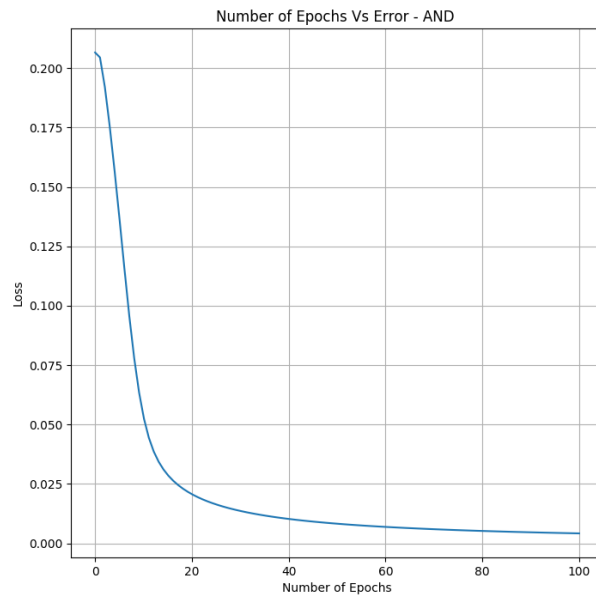


(a) AND

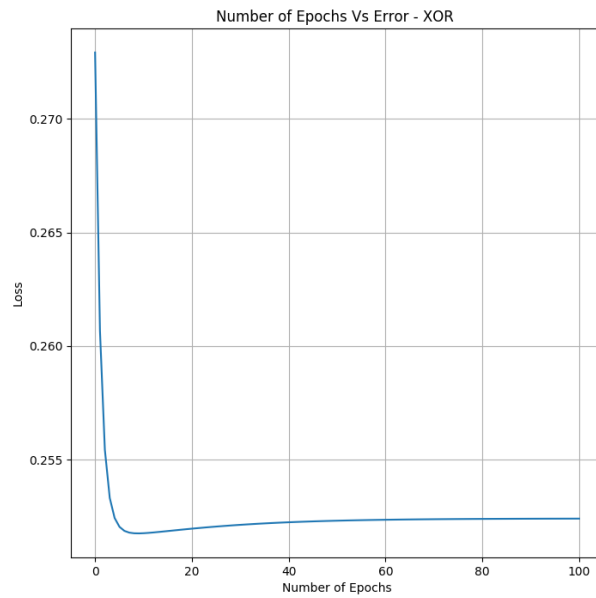


(b) XOR

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

```
1  '''
2  Created by: Kevin De Angeli
3  Email: kevindeangeli@utk.edu
4  Date: 2020-01-15
5
6  Note: The activation and loss function are in two separate files
7  which will be included in the zip file.
8
9  General description:
10 The NeuralNetwork objects will contain objects of the FullyConnectedLayer class.
11 The FullyConnectedLayer contains objects of the Neuron class. Each of the Neuron
12 objects contains their own weights, biase, and delta value.
13
14 I created a method in the Neuron function called "mini_Delta" which is used to
15 compute the weight (given a certain index) times the delta of that neuron. This
16 is used for backpropagation. The backpropagation function in the NeuralNetwork
17     ↳ first
18 computes the Sum Delta values for the layers, and these are passed to individual
19 layers so that the weights of each neuron can be updated.
20
21 '''
22
23 from errorFunctions import * #I put the error functions and their derivatives in a
24     ↳ different file.
25 from activation import *      #I put the activation functions and their derivatives
26     ↳ in a different file.
27
28 import numpy as np
29 import sys
30 import matplotlib.pyplot as plt
31
32 class Neuron():
33
34     def __init__(self, inputLen, activationFun = "sigmoid", lossFunction="mse" ,
35         ↳ learningRate = .5, weights = None, bias = None):
36         self.inputLen = inputLen
37         self.learnR = learningRate
38         self.activationFunction = activationFun
39         self.lossFunction = lossFunction
```

```

38     self.output = None #Saving the output of the neuron (after feedforward)
39         ↳ makes things easy for backprop
40     self.input = None #Saves the input to the neuron for backprop.
41     self.newWeights = [] #Saves new weight but it doesn't update until the end
42         ↳ of backprop. (method: updateWeight)
43     self.newBias = None
44     self.delta = None #individual deltas required for backprop.
45
46     if weights is None:
47         #set them to random:
48         self.weights = [np.random.random_sample() for i in range(inputLen)]
49         self.bias = np.random.random_sample()
50
51     else:
52         self.weights = weights
53         self.bias = bias
54
55     #this series of if statement define the activation and loss functions, and
56     ↳ their derivatives.
57     if activationFun is "sigmoid":
58         self.activate = sigmoid
59         self.activation_prime = sigmoid_prime
60     else:
61         self.activate = linear
62
63     if lossFunction is "mse":
64         self.loss = mse
65         self.loss_prime = mse_prime
66     else:
67         self.loss = crossEntropy
68         self.loss_prime = crossEntropy_prime
69
70     #The following pictures will be defined based on the parameters
71     #that is passed to the object.
72     def activate(self):
73         pass
74     def loss(self):
75         pass
76     def activation_prime(self):
77         pass
78     def loss_prime(self):
79         pass
80
81     #This function is called after backpropagation.
82     def updateWeight(self):
83         self.weights = self.newWeights

```

```

81     self.newWeights = []
82     self.bias = self.newBias
83     self.newBias = None
84
85     def calculate(self, input):
86         '''
87         Given an input, it will calculate the output
88         :return:
89         '''
90         self.input = input
91         self.output = self.activate(np.dot(input, self.weights) + self.bias)
92         return self.output
93
94     #The delta of the last layer is computed a little different, so it has its own
95     ↳ function.
96     def backpropagationLastLayer(self, target):
97         self.delta = self.loss_prime(self.output, target) *
98         ↳ self.activation_prime(self.output)
99         self.newBias = self.bias - self.learnR * self.delta
100         for index, PreviousNeuronOutput in enumerate(self.input):
101             self.newWeights.append(self.weights[index] - self.learnR * self.delta *
102             ↳ PreviousNeuronOutput)
103
104     def backpropagation(self, sumDelta):
105         #sumDelta will be computed at the layer level. Since it requires weights
106         ↳ from multiple neurons.
107         self.delta = sumDelta * self.activation_prime(self.output)
108         self.newBias = self.bias - self.learnR * self.delta
109         for index, PreviousNeuronOutput in enumerate(self.input):
110             self.newWeights.append(self.weights[index] - self.learnR * self.delta *
111             ↳ self.input[index])
112
113     #Used to compute the sumation of the Deltas for backprop.
114     def mini_Delta(self, index):
115         return self.delta * self.weights[index]
116
117 class FullyConnectedLayer():
118     def __init__(self, inputLen, numOfNeurons = 5, activationFun = "sigmoid",
119     ↳ lossFunction= "mse", learningRate = .1, weights = None, bias = None):
120         self.inputLen = inputLen
121         self.neuronsNum = numOfNeurons
122         self.activationFun = activationFun
123         self.learningRate = learningRate
124         self.weights = weights

```

```

121     self.bias = bias
122     self.layerOutput = []
123     self.lossFunction = lossFunction
124
125     #Random weights or user defined weights:
126     if weights is None:
127         self.neurons = [Neuron(inputLen=self.inputLen,
128             ↪ activationFun=activationFun,lossFunction=self.lossFunction
129             ↪ ,learningRate=self.learningRate, weights=self.weights) for i in
130             ↪ range(numOfNeurons)]
131     else:
132         self.neurons = [Neuron(inputLen=self.inputLen,
133             ↪ activationFun=activationFun,lossFunction=self.lossFunction,
134             ↪ learningRate=self.learningRate, weights=self.weights[i], bias=
135             ↪ self.bias[i]) for i in range(numOfNeurons)]
136
137 def calculate(self, input):
138     '''
139     Will calculate the output of all the neurons in the layer.
140     :return:
141     '''
142     self.layerOutput = []
143     for neuron in self.neurons:
144         self.layerOutput.append(neuron.calculate(input))
145
146     return self.layerOutput
147
148 def backPropagateLast(self, target):
149     for targetIndex, neuron in enumerate(self.neurons):
150         neuron.backpropagationLastLayer(target=target[targetIndex])
151
152 def updateWeights(self):
153     for neuron in self.neurons:
154         neuron.updateWeight()
155
156 #Computes the sum of the deltas times their weights based on the number of
157 ↪ neurons in the previous layer.
158 def deltaSum(self):
159     delta_sumArr = []
160     x=len(self.neurons[0].weights)
161     for i in range(len(self.neurons[0].weights)): #Number of Weights in the
162         ↪ RightLayer = Number of neurons in the LeftLayer
163         delta_sum = 0
164         for index, neuron in enumerate(self.neurons):
165             delta_sum += neuron.mini_Delta(i)

```



```

159         delta_sumArr.append(delta_sum)
160     return delta_sumArr
161
162     def backpropagation(self, deltaArr):
163         #Each neuron needs a delta to update their weights:
164         for index, neuron in enumerate(self.neurons):
165             neuron.backpropagation(deltaArr[index])
166
167
168 class NeuralNetwork():
169     def __init__(self, neuronsNum = None, activationVector = 0, lossFunction =
170         ↪ "mse", learningRate = .1, weights = None, bias = None):
171         self.inputLen = neuronsNum[0]
172         self.layersNum = len(neuronsNum)-1 #Don't count the first one (input).
173         self.activationVector = activationVector
174         self.lossFunction = lossFunction
175         self.learningRate = learningRate
176         self.weights = weights
177         self.bias = bias
178
179         if neuronsNum is None : #By default, each layer will have 3 neurons,
180             ↪ unless specified.
181             self.neuronsNum = [3 for i in range(layersNum)]
182         else:
183             self.neuronsNum = neuronsNum[1:len(neuronsNum)] #Don't count the first
184             ↪ one (input)
185
186         if activationVector is None or activationVector != self.layersNum: #This is
187             ↪ the default vector if one is not provided when the class is created.
188             self.activationVector = ["sigmoid" for i in range(self.layersNum)]
189
190         #Define the layers of the networks with the respective neurons:
191         self.layers = []
192         inputLenLayer = self.inputLen
193         #This convoluted loop creates the layers and neurons with the appropite
194         ↪ number of weights in each.
195         if weights is None:
196             for i in range(self.layersNum):
197                 self.layers.append(
198                     FullyConnectedLayer(numOfNeurons=self.neuronsNum[i],
199                     ↪ activationFun=self.activationVector[i],lossFunction=self.lossFunction,
200                     ↪ inputLen=inputLenLayer, learningRate=self.learningRate,
201                     ↪ weights=self.weights))
202
203                 # The number of weights in one layer depends on the number of
204                 ↪ neurons in the previous layer:
205                 inputLenLayer = self.neuronsNum[i]

```

```

196         #Used defined weights:
197     else:
198         for i in range(self.layersNum):
199             self.layers.append(
200                 FullyConnectedLayer(numOfNeurons=self.neuronsNum[i],
201                                     activationFun=self.activationVector[i],
202                                     inputLen=inputLenLayer, learningRate=self.learningRate,
203                                     weights=self.weights[i], bias=self.bias[i]))
204             # The number of weights in one layer depends on the number of
205             neurons in the previous layer:
206             inputLenLayer = self.neuronsNum[i]
207
208     def showWeights(self):
209         #Function which just goes through each neuron in each layer and displays
210         the weights.
211         inputLenLayer = self.inputLen
212         for i in range(self.layersNum):
213             print(" ")
214             for k in range(self.neuronsNum[i]):
215                 print(self.layers[i].neurons[k].weights)
216
217             inputLenLayer = self.neuronsNum[i]
218
219     def showBias(self):
220         #Function which just goes through each neuron in each layer and displays
221         the bias.
222         inputLenLayer = self.inputLen
223         for i in range(self.layersNum):
224             #print(" ")
225             for k in range(self.neuronsNum[i]):
226                 print(self.layers[i].neurons[k].bias)
227
228             inputLenLayer = self.neuronsNum[i]
229
230     def calculate(self, input):
231         '''
232         given an input calculates the output of the network.
233         input should be a list.
234         :return:
235         '''
236         output = input
237         for layer in self.layers:
238             output = layer.calculate(output)

```

```

236
237     return output
238
239 def backPropagate(self, target):
240     self.layers[-1].backPropagateLast(target)
241     layersCounter = self.layersNum+1
242
243     for i in range(2,layersCounter):
244         #Calculate the sum delta for the following layer to update the previous
245             layer.
246         deltaArr = self.layers[-i + 1].deltaSum()
247         self.layers[-i].backpropagation(deltaArr)
248
249     for layer in self.layers:
250         layer.updateWeights()
251
252
253 def calculateLoss(self,input,target, function = "mse"):
254     '''
255     Given an input and desired output, calculate the loss.
256     Can be implemented with MSE and binary cross.
257     '''
258     N = len(input)
259     output = self.calculate(input)
260     if function == "mse":
261         error = mse(output, target)
262     else:
263         crossEntropy(output, target)
264
265     return error
266
267
268 def train(self, input, target, showLoss = False):
269     '''
270     Basically, do forward and backpropagation all together here.
271     Given a single input and desired output, it will take one step of gradient
272         descent.
273     :return:
274     '''
275     self.calculate(input)
276     if showLoss is True:
277         print("mse: ", self.calculateLoss(input=input, target=target))
278     self.backPropagate(target)
279

```

```

280 def doExample():
281     '''
282     This function does the "Example" forward and backpop pass required for the
    ↪ assignment.
283     '''
284     print( "--- Example ---")
285
286     #Let's try the class example by setting the bias and weights:
287     Newweights = [[ [.15,.20], [.25, .30]], [[.40, .45], [.5, .55]]]
288     newBias = [[.35,.35],[.6,.6]]
289     model = NeuralNetwork(neuronsNum=[2, 2, 2], activationVector=['sigmoid',
    ↪ 'sigmoid'], lossFunction="mse",
290                             learningRate=.5, weights=Newweights, bias = newBias)
291
292
293
294     print("Original weights and biases of the network: ")
295     print("Model's Weights:")
296     model.showWeights()
297     print("\nModel's Bias:")
298     model.showBias()
299
300
301     print("\nForward pass: ")
302     print(model.calculate([.05,.1]))
303
304     #model.train(input= [.05,.1], target=[.01, .99]) #you could use just this
    ↪ function to do all at once.
305     model.backPropagate(target= [.01, .99])
306     print("\nAfter BackProp, the updated weights are:")
307     print("Model's Weights:")
308     model.showWeights()
309     print("\nModel's Bias:")
310     model.showBias()
311
312 def doAnd():
313     '''
314     This function trains a single neuron to learn the "AND" logical operator.
315     '''
316     print( "\n--- AND ---")
317     x = [[1,1],[1,0],[0,1], [0,0]]
318     y = [[1],[0], [0], [0]]
319
320     model = NeuralNetwork(neuronsNum=[2, 1], activationVector=['sigmoid'],
    ↪ lossFunction="mse",
321                             learningRate=6, weights=None, bias=None)

```

```

322
323
324 print("----- Before training -----")
325 print("Model's Weights:")
326 model.showWeights()
327 print("\nModel's Bias:")
328 model.showBias()
329
330
331
332 for i in range(10000):
333     for index in range(len(x)):
334         model.train(input=x[index],target=y[index])
335
336 print("-----")
337 print("\nPredictions: ")
338 for index2 in range(len(x)):
339     print("\nPredict: ", x[index2])
340     print(model.calculate(x[index2]))
341
342 print("----- After training -----")
343 print("Model's Weights:")
344 model.showWeights()
345 print("\nModel's Bias:")
346 model.showBias()
347
348
349
350
351 def doXor():
352     '''
353     This function creates two models: 1) A single neuron model which is not able to
354     learn the XOR operator, and 2) A model with two neurons in the hidden layer,
355     and 1 output neuron which successfully learns XOR.
356     '''
357     print( "\n--- XOR ---")
358     x = [[1,1],[1,0],[0,1], [0,0]]
359     y = [[0],[1], [1], [0]]
360
361     print("First model: [2,1] (Single Perceptron) : \n")
362     model = NeuralNetwork(neuronsNum=[2, 1], activationVector=['sigmoid'],
363         lossFunction="mse",
364         learningRate=6, weights=None, bias=None)
365
366     print("----- Before training -----")
367     print("Model's Weights:")

```

```

367 model.showWeights()
368 print("\nModel's Bias:")
369 model.showBias()
370
371 for i in range(10000):
372     for index in range(len(x)):
373         model.train(input=x[index], target=y[index])
374
375 print("-----")
376 print("\nPredictions: ")
377 for index2 in range(len(x)):
378     print("\nPredict: ", x[index2])
379     print(model.calculate(x[index2]))
380
381 print("----- After training -----")
382 print("Model's Weights:")
383 model.showWeights()
384 print("\nModel's Bias:")
385 model.showBias()
386 print("\n\n *****\n\n")
387 print("Second model: [2,2,1] (Single Perceptron) : \n")
388
389 model = NeuralNetwork(neuronsNum=[2, 2, 1], activationVector=['sigmoid',
390     'sigmoid'], lossFunction="mse",
391     learningRate=.5, weights=None, bias=None)
392
393 print("----- Before training -----")
394 print("Model's Weights:")
395 model.showWeights()
396 print("\nModel's Bias:")
397 model.showBias()
398
399 for i in range(10000): #It works with 100000 and alpha = 1.5 but it takes a
400     minute
401     for index in range(len(x)):
402         model.train(input=x[index], target=y[index])
403
404 print("-----")
405 print("\nPredictions: ")
406 for index2 in range(len(x)):
407     print("\nPredict: ", x[index2])
408     print(model.calculate(x[index2]))
409
410 print("----- After training -----")
411 print("Model's Weights:")
412 model.showWeights()

```

```

411 print("\nModel's Bias:")
412 model.showBias()
413
414
415 def showLoss(learningRate, data = "and"):
416     '''
417     This function creates the Learnign Rate vs Loss plot for both: AND and XOR.
418     '''
419     if data is "and":
420         print("Loss for AND")
421         x = [[1,1],[1,0],[0,1], [0,0]]
422         y = [[1],[0], [0], [0]]
423         title = "LearningRateVsErrorAND_MLE.png"
424         figTitle= "Learning Rate Vs Error - AND "
425
426     else:
427         print("Loss for XOR")
428         x = [[1, 1], [1, 0], [0, 1], [0, 0]]
429         y = [[0], [1], [1], [0]]
430         title = "LearningRateVsErrorXOR_MLE.png"
431         figTitle= "Learning Rate Vs Error - XOR "
432
433
434 errorAvrage = [] #This will contain the Ys of the plot.
435 for i in learningRate:
436     model = NeuralNetwork(neuronsNum=[2, 1], activationVector=['sigmoid'],
437         ↪ lossFunction="mse",
438         learningRate=i, weights=None, bias=None)
439
440     errorListPerLearningRate = []
441     for i in range(10):
442         errorList = []
443         for index in range(len(x)): #Train the algorithm with entire dataset
444             model.train(input=x[index], target=y[index])
445
446         for index2 in range(len(x)):
447             errorList.append(model.calculateLoss(input=x[index2],
448                 ↪ target=y[index2])) #Collect individual errors
449
450         errorListPerLearningRate.append(np.average(errorList))
451
452     errorAvrage.append(np.average(errorListPerLearningRate))
453
454 fig, ax = plt.subplots(figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
455 ax.plot(learningRate, errorAvrage)
456 ax.set(xlabel='Learning Rate', ylabel='Loss',title=figTitle)

```

```

455     ax.grid()
456     plt.savefig(title)
457     plt.show()
458
459     def lossVSEpoch(data="and"):
460         '''
461         This function creates the plots that displays the loss as a function of the
462         number of epochs.
463         '''
464         if data is "and":
465             print("Loss for AND")
466             x = [[1,1],[1,0],[0,1], [0,0]]
467             y = [[1],[0], [0], [0]]
468             title = "EpochVsErrorAND_MLE.png"
469             figTitle= "Number of Epochs Vs Error - AND "
470             learnRate = 5.5
471
472         else:
473             print("Loss for XOR")
474             x = [[1, 1], [1, 0], [0, 1], [0, 0]]
475             y = [[0], [1], [1], [0]]
476             title = "EpochVsErrorXOR_MLE.png"
477             figTitle= "Number of Epochs Vs Error - XOR "
478             learnRate = 2
479
480     errorArr = [] #This will contain the Ys of the plot.
481
482     model = NeuralNetwork(neuronsNum=[2, 1], activationVector=['sigmoid'],
483         lossFunction="mse",
484         learningRate=learnRate, weights=None, bias=None)
485     epochsNums = 100
486     for i in range(epochsNums):
487         errorList = []
488         for index in range(len(x)): #Train the algorithm with entire dataset
489             model.train(input=x[index], target=y[index])
490
491         for index2 in range(len(x)):
492             errorList.append(model.calculateLoss(input=x[index2],
493                 target=y[index2])) #Collect individual errors
494
495         errorArr.append(np.average(errorList))
496
497     fig, ax = plt.subplots(figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
498     ax.plot(np.linspace(0,epochsNums,epochsNums), errorArr)

```



```

498     ax.set(xlabel='Number of Epochs', ylabel='Loss',title=figTitle)
499     ax.grid()
500     plt.savefig(title)
501     plt.show()
502
503
504 def main():
505     program_name = sys.argv[0]
506     input = sys.argv[1:] #Get input from the console.
507     # Input validation:
508     if len(input) != 1:
509         print("Input only one of these: example, and, or xor")
510         return 0
511
512     # This is just to run it from the editor instead of the console.
513     #input = ["example", "and", "xor"]
514     #userInput = input[2]
515
516     if input[0] == "example":
517         doExample()
518     elif input[0] == "and":
519         doAnd()
520     elif input[0] == "xor":
521         doXor()
522     elif input[0] == "lossLearning":
523         learningRateArr = np.linspace(0.1, 12, num=50)
524         showLoss(learningRateArr, data="and")
525     elif input[0] == "lossEpoch":
526         lossVSEpoch(data="xor")
527     else:
528         # Input validation
529         print("Input Options: example, and, or xor")
530         return 0
531
532 if __name__ == "__main__":
533     main()

```

6.1.2 activation.py

```

1  '''
2  Created by: Kevin De Angeli
3  Email: kevindeangeli@utk.edu
4  Date: 2020-01-16
5
6  '''
7
8  import numpy as np

```

```

9
10
11 def sigmoid(x):
12     return 1 / (1 + np.exp(-x))
13
14 def sigmoid_prime(z):
15     """Derivative of the sigmoid function."""
16     x = z * (1 - z)
17     return z * (1 - z)
18
19 def linear(x):
20     return x

```

6.1.3 errorFunctions.py

```

1  '''
2  Created by: Kevin De Angeli
3  Email: kevindeangeli@utk.edu
4  Date: 2020-01-16
5
6  '''
7
8  import numpy as np
9
10 def crossEntropy(prediction, output):
11     if output == 1:
12         return -log(prediction)
13     else:
14         return -log(1 - prediction)
15
16 def crossEntropy_prime(prediction, output):
17     return -((output/prediction) - ((1-output)/(1-prediction)))
18
19 def mse(prediction, output):
20     if len(prediction) == 1:
21         return (prediction[0]-output[0])**2
22     else:
23         return ((prediction - output)**2).mean(axis=0)
24
25 def mse_prime(output, target):
26     return output-target

```