
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 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

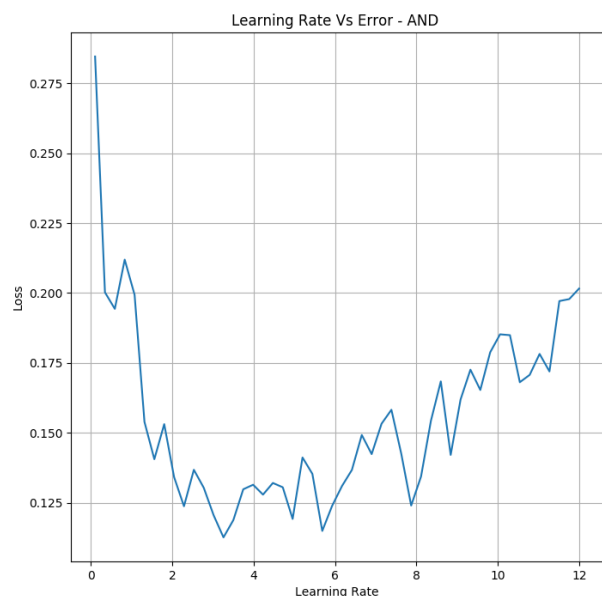
4 Running the Code: Instructions

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.

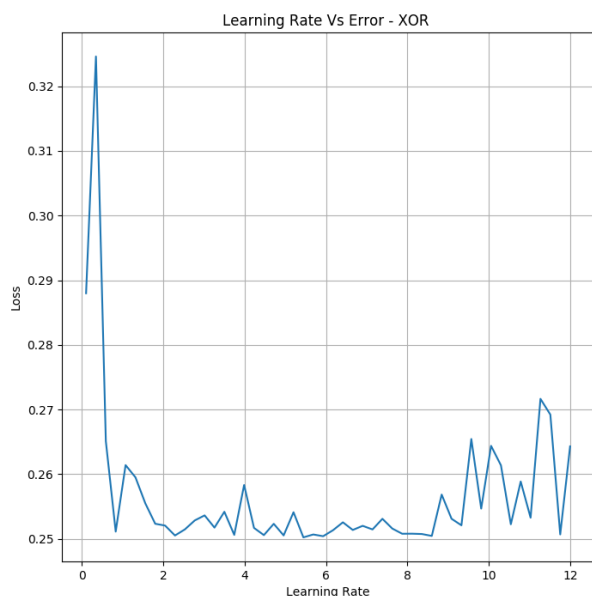
5 Loss plots

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).

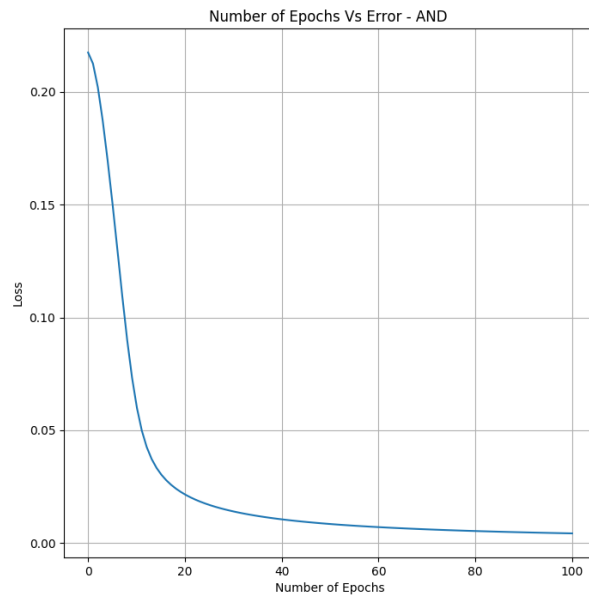


(a) AND

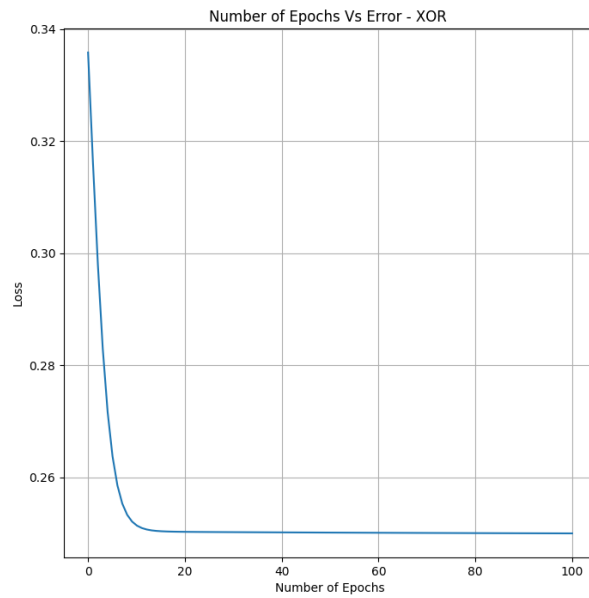


(b) XOR

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         x1= self.loss_prime(self.output, target)
100        x2= self.activation_prime(self.output)
101        self.newBias = self.bias - self.learnR*self.delta
102        for index, PreviousNeuronOutput in enumerate(self.input):
103            self.newWeights.append(self.weights[index] - self.learnR * self.delta *
104            ↳ PreviousNeuronOutput)
105
106    def backpropagation(self, sumDelta):
107        #sumDelta will be computed at the layer level. Since it requires weights
108        ↳ from multiple neurons.
109        self.delta = sumDelta * self.activation_prime(self.output)
110        self.newBias = self.bias - self.learnR * self.delta
111        for index, PreviousNeuronOutput in enumerate(self.input):
112            self.newWeights.append(self.weights[index] - self.learnR * self.delta *
113            ↳ self.input[index])
114
115    #Used to compute the sumation of the Deltas for backprop.
116    def mini_Delta(self, index):
117        return self.delta * self.weights[index]
118
119    class FullyConnectedLayer():
120        def __init__(self, inputLen, numOfNeurons = 5, activationFun = "sigmoid",
121            ↳ lossFunction= "mse", learningRate = .1, weights = None, bias = None):
122            self.inputLen = inputLen
123            self.neuronsNum = numOfNeurons
124            self.activationFun = activationFun

```

```

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

```



```

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

```

```

196         # The number of weights in one layer depends on the number of
197         ↳ neurons in the previous layer:
198
199         #Used defined weights:
200     else:
201         for i in range(self.layersNum):
202             self.layers.append(
203                 FullyConnectedLayer(numOfNeurons=self.neuronsNum[i],
204                                     ↳ activationFun=self.activationVector[i],
205                                     ↳ inputLen=inputLenLayer, learningRate=self.learningRate,
206                                     ↳ weights=self.weights[i], bias=self.bias[i]))
207
208         # The number of weights in one layer depends on the number of
209         ↳ neurons in the previous layer:
210         inputLenLayer = self.neuronsNum[i]
211
212
213 def showWeights(self):
214     #Function which just goes through each neuron in each layer and displays
215     ↳ the weights.
216     inputLenLayer = self.inputLen
217     for i in range(self.layersNum):
218         print(" ")
219         for k in range(self.neuronsNum[i]):
220             print(self.layers[i].neurons[k].weights)
221
222     inputLenLayer = self.neuronsNum[i]
223
224
225 def showBias(self):
226     #Function which just goes through each neuron in each layer and displays
227     ↳ the bias.
228     inputLenLayer = self.inputLen
229     for i in range(self.layersNum):
230         #print(" ")
231         for k in range(self.neuronsNum[i]):
232             print(self.layers[i].neurons[k].bias)
233
234     inputLenLayer = self.neuronsNum[i]
235
236
237 def calculate(self, input):
238     '''
239     given an input calculates the output of the network.
240     input should be a list.
241     :return:
242     '''

```

```

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

```

```

279         self.backPropagate(target)
280
281
282 def doExample():
283     '''
284     This function does the "Example" forward and backpop pass required for the
    ↪ assignemnt.
285     '''
286     print( "--- Example ---")
287
288     #Let's try the class example by setting the bias and weights:
289     Newweights = [[.15,.20], [.25, .30]], [[.40, .45], [.5, .55]]
290     newBias = [[.35,.35],[.6,.6]]
291     model = NeuralNetwork(neuronsNum=[2, 2, 2], activationVector=['sigmoid',
    ↪ 'sigmoid'], lossFunction="mse",
292                             learningRate=.5, weights=Newweights, bias = newBias)
293
294
295
296     print("Original weights and biases of the network: ")
297     print("Model's Weights:")
298     model.showWeights()
299     print("\nModel's Bias:")
300     model.showBias()
301
302
303     print("\nForward pass: ")
304     print(model.calculate([.05,.1]))
305
306     #model.train(input= [.05,.1], target=[.01, .99]) #you could use just this
    ↪ function to do all at once.
307     model.backPropagate(target= [.01, .99])
308     print("\nAfter BackProp, the updated weights are:")
309     print("Model's Weights:")
310     model.showWeights()
311     print("\nModel's Bias:")
312     model.showBias()
313
314 def doAnd():
315     '''
316     This function trains a single neuron to learn the "AND" logical operator.
317     '''
318     print( "\n--- AND ---")
319     x = [[1,1],[1,0],[0,1], [0,0]]
320     y = [[1],[0], [0], [0]]
321

```

```

322 model = NeuralNetwork(neuronsNum=[2, 1], activationVector=['sigmoid'],
323     ↪ lossFunction="mse",
324     learningRate=6, weights=None, bias=None)
325
326 print("----- Before training -----")
327 print("Model's Weights:")
328 model.showWeights()
329 print("\nModel's Bias:")
330 model.showBias()
331
332
333
334 for i in range(10000):
335     for index in range(len(x)):
336         model.train(input=x[index],target=y[index])
337
338 print("-----")
339 print("\nPredictions: ")
340 for index2 in range(len(x)):
341     print("\nPredict: ", x[index2])
342     print(model.calculate(x[index2]))
343
344 print("----- After training -----")
345 print("Model's Weights:")
346 model.showWeights()
347 print("\nModel's Bias:")
348 model.showBias()
349
350
351
352
353 def doXor():
354     '''
355     This function creates two models: 1) A single neuron model which is not able to
356     learn the XOR operator, and 2) A model with two neurons in the hidden layer,
357     and 1 output neuron which successfully learns XOR.
358     '''
359     print( "\n--- XOR ---")
360     x = [[1,1],[1,0],[0,1], [0,0]]
361     y = [[0],[1], [1], [0]]
362
363     print("First model: [2,1] (Single Perceptron) : \n")
364     model = NeuralNetwork(neuronsNum=[2, 1], activationVector=['sigmoid'],
365     ↪ lossFunction="mse",
366     learningRate=1.9, weights=None, bias=None)

```

```

366
367 print("----- Before training -----")
368 print("Model's Weights:")
369 model.showWeights()
370 print("\nModel's Bias:")
371 model.showBias()
372
373 for i in range(10000):
374     for index in range(len(x)):
375         model.train(input=x[index], target=y[index])
376
377 print("-----")
378 print("\nPredictions: ")
379 for index2 in range(len(x)):
380     print("\nPredict: ", x[index2])
381     print(model.calculate(x[index2]))
382
383 print("----- After training -----")
384 print("Model's Weights:")
385 model.showWeights()
386 print("\nModel's Bias:")
387 model.showBias()
388 print("\n\n *****\n\n")
389 print("Second model: [2,2,1] (Hidden layer Perceptron) : \n")
390
391 model = NeuralNetwork(neuronsNum=[2, 2, 1], activationVector=['sigmoid',
392     'sigmoid'], lossFunction="mse",
393     learningRate=.5, weights=None, bias=None)
394
395 print("----- Before training -----")
396 print("Model's Weights:")
397 model.showWeights()
398 print("\nModel's Bias:")
399 model.showBias()
400
401 for i in range(10000): #It works with 100000 and alpha = 1.5 but it takes a
402     minute
403     for index in range(len(x)):
404         model.train(input=x[index], target=y[index])
405
406 print("-----")
407 print("\nPredictions: ")
408 for index2 in range(len(x)):
409     print("\nPredict: ", x[index2])
410     print(model.calculate(x[index2]))

```

```

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

```

```

454         errorAvrage.append(np.average(errorListPerLearningRate))
455
456     fig, ax = plt.subplots(figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
457     ax.plot(learningRate, errorAvrage)
458     ax.set(xlabel='Learning Rate', ylabel='Loss', title=figTitle)
459     ax.grid()
460     plt.savefig(title)
461     plt.show()
462
463     def lossVSEpoch(data="and"):
464         '''
465         This function creates the plots that displays the loss as a function of the
466         number of epochs.
467         '''
468         if data is "and":
469             print("Loss for AND")
470             x = [[1,1],[1,0],[0,1], [0,0]]
471             y = [[1],[0], [0], [0]]
472             title = "EpochVsErrorAND_MLE.png"
473             figTitle= "Number of Epochs Vs Error - AND "
474             learnRate = 5.5
475             modelArch = [2,1]
476
477         else:
478             print("Loss for XOR")
479             x = [[1, 1], [1, 0], [0, 1], [0, 0]]
480             y = [[0], [1], [1], [0]]
481             title = "EpochVsErrorXOR_MLE.png"
482             figTitle= "Number of Epochs Vs Error - XOR "
483             learnRate = .5
484             modelArch = [2,2,1]
485
486     errorArr = [] #This will contain the Ys of the plot.
487
488     model = NeuralNetwork(neuronsNum=modelArch, activationVector=None,
489         lossFunction="mse",
490         learningRate=learnRate, weights=None, bias=None)
491     epochsNums = 100
492     for i in range(epochsNums):
493         errorList = []
494         for index in range(len(x)): #Train the algorithm with entire dataset
495             model.train(input=x[index], target=y[index])
496
497         for index2 in range(len(x)):

```



```

498         errorList.append(model.calculateLoss(input=x[index2],
499         ~ target=y[index2])) #Collect individual errors
500
501     errorArr.append(np.average(errorList))
502
503     fig, ax = plt.subplots(figsize=(8, 8), dpi=100, facecolor='w', edgecolor='k')
504     ax.plot(np.linspace(0,epochsNums,epochsNums), errorArr)
505     ax.set(xlabel='Number of Epochs', ylabel='Loss',title=figTitle)
506     ax.grid()
507     plt.savefig(title)
508     plt.show()
509
510
511 def main():
512     program_name = sys.argv[0]
513     #input = sys.argv[1:] #Get input from the console.
514     # Input validation:
515     #if len(input) != 1:
516     #     print("Input only one of these: example, and, or xor")
517     #     return 0
518
519     # This is just to run it from the editor instead of the console.
520     input = ["example", "and", "xor","lossLearning", "lossEpoch"]
521     input = [input[3]]
522
523     if input[0] == "example":
524         doExample()
525     elif input[0] == "and":
526         doAnd()
527     elif input[0] == "xor":
528         doXor()
529     elif input[0] == "lossLearning":
530         learningRateArr = np.linspace(0.1, 12, num=50)
531         showLoss(learningRateArr, data="xor")
532     elif input[0] == "lossEpoch":
533         lossVSEpoch(data="xor")
534     else:
535         # Input validation
536         print("Input Options: example, and, or xor")
537         return 0
538
539
540
541 if __name__ == "__main__":
542     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
```