/**[WineNeuralNet](https://github.com/JohnNov/WineNeuralNet)**

[**WineNeuralNet**](https://github.com/JohnNov/WineNeuralNet)**/wineNeuralNet.py**

[**Find file**](https://github.com/JohnNov/WineNeuralNet/find/master)Copy path

[@JohnNov](https://github.com/JohnNov) [JohnNov](https://github.com/JohnNov) [Second commit3644792](https://github.com/JohnNov/WineNeuralNet/commit/36447921db0423b55bd6f17f93fbc13e502ecc65) on Apr 4, 2017

**1** contributor

169 lines (118 sloc)  5.27 KB

[**Raw**](https://github.com/JohnNov/WineNeuralNet/raw/master/wineNeuralNet.py)[**Blame**](https://github.com/JohnNov/WineNeuralNet/blame/master/wineNeuralNet.py)[**History**](https://github.com/JohnNov/WineNeuralNet/commits/master/wineNeuralNet.py)

|  |  |
| --- | --- |
|  | import numpy as np |
|  | import math |
|  | import matplotlib.pyplot as plt |
|  | from numpy import linalg as LA |
|  | from scipy import optimize |
|  | ######################################################################## |
|  | #We've got 50 values from each. Then we scale each of them |
|  | #to a range of [0,1] |
|  | ######################################################################## |
|  | def prepAndNormalizeData(string): |
|  | anArray = [] |
|  |  |
|  | with open(string+".txt") as f: |
|  | for line in f: |
|  |  |
|  | line = line.strip('\n') |
|  | line = line.strip('') |
|  | line = float(line) |
|  | anArray.append(line) |
|  |  |
|  |  |
|  | newArray = np.array(anArray) |
|  | newArray = newArray-min(anArray) |
|  | newArray = newArray/(max(newArray)-min(newArray)) |
|  | f.close() |
|  | return newArray |
|  |  |
|  | sugar = prepAndNormalizeData("sugar") |
|  |  |
|  | density = prepAndNormalizeData("density") |
|  |  |
|  | alcohol = prepAndNormalizeData("alcohol") |
|  |  |
|  | y = prepAndNormalizeData("quality") |
|  |  |
|  | rows = 49 |
|  |  |
|  | cols = 3 |
|  |  |
|  | X = np.zeros((50,3)) #input matrix X = (sugar, density, alcohol). Should be 3 col and 50 rows |
|  |  |
|  | for i in range(50): |
|  | X[i][0] = (sugar[i]) |
|  | X[i][1] = (density[i]) |
|  | X[i][2] = (alcohol[i]) |
|  |  |
|  | Y = np.zeros((50,1)) |
|  |  |
|  | for i in range(50): |
|  | Y[i][0] = (y[i]) |
|  |  |
|  | y = Y |
|  |  |
|  | ######################################################################## |
|  | #Now we create the actual structure/hyperparameters of the neural net |
|  | ######################################################################## |
|  |  |
|  | class neuralNetwork(): |
|  |  |
|  | def \_\_init\_\_(self): |
|  | self.inputLayerSize = 3 |
|  | self.outputLayerSize = 1 |
|  | self.hiddenLayerSize = 4 |
|  | self.W1 = np.random.randn(3, 4)\*math.sqrt(2.0/3) #Neurons at this layer get 3 inputs |
|  | self.W2 = np.random.randn(4, 1)\*math.sqrt(2.0/4) |
|  |  |
|  |  |
|  | def relu(self, z): #Additional/optional activation function. Performs poorly at the moment |
|  | return np.maximum(0, z) |
|  |  |
|  | def reluPrime(self,z): #Necessary derivative for gradient calculation |
|  | z = (z>0) #Converts all z2 elements to either TRUE or FALSE if element greater than 0 |
|  | z = z+0 |
|  | return z |
|  |  |
|  | def sigmoid(self, z): #Apply sigmoid activation function to scalar, vector, or matrix |
|  | return 1/(1+np.exp(-z)) |
|  |  |
|  | def sigmoidPrime(self,z): #Gradient of sigmoid |
|  | return np.exp(-z)/((1+np.exp(-z))\*\*2) |
|  |  |
|  | def forward(self, X): |
|  | self.z2 = np.dot(X,self.W1) #Second layer activity, hence z2 |
|  | self.a2 = self.sigmoid(self.z2) |
|  | self.z3 = np.dot(self.a2,self.W2) #Third layer activity, hence z3 |
|  | yHat = self.sigmoid(self.z3) |
|  |  |
|  | return yHat #Our neural network's "guess" |
|  |  |
|  |  |
|  | #Benefit of Quadratic Cost Function is that it limits the times we run into a local minimum problem (Though not always) |
|  |  |
|  | def costFunction(self, X, y): |
|  | self.yHat = self.forward(X) |
|  | L = 0.5\*sum((y-self.yHat)\*\*2) |
|  | return L |
|  |  |
|  |  |
|  | #Compute derivative with respect to W and W2 for a given X and y: |
|  | #See http://neuralnetworksanddeeplearning.com/ for thorough understanding of |
|  | #How to derive these gradients. We essentially just use the chain rule a lot. |
|  |  |
|  | def costFunctionPrime(self, X, y): |
|  |  |
|  | self.yHat = self.forward(X) |
|  | dLdyHat = np.subtract(yHat,y) |
|  | delta3 = np.multiply((dLdyHat), self.sigmoidPrime(self.z3)) |
|  | dJdW2 = np.dot(self.a2.T, delta3) |
|  |  |
|  | delta2 = np.dot(delta3, self.W2.T)\*self.sigmoidPrime(self.z2) |
|  | dJdW1 = np.dot(X.T, delta2) |
|  | return dJdW1, dJdW2 |
|  |  |
|  | def getParams(self): |
|  | #Get W1 and W2 unrolled into vector: |
|  | params = np.concatenate((self.W1.ravel(), self.W2.ravel())) |
|  | return params |
|  |  |
|  | def setParams(self, params): |
|  | #Set W1 and W2 using single paramater vector. |
|  | W1\_start = 0 |
|  | W1\_end = self.hiddenLayerSize \* self.inputLayerSize |
|  | self.W1 = np.reshape(params[W1\_start:W1\_end], (self.inputLayerSize , self.hiddenLayerSize)) |
|  | W2\_end = W1\_end + self.hiddenLayerSize\*self.outputLayerSize |
|  | self.W2 = np.reshape(params[W1\_end:W2\_end], (self.hiddenLayerSize, self.outputLayerSize)) |
|  |  |
|  | def computeGradients(self, X, y): |
|  | dJdW1, dJdW2 = self.costFunctionPrime(X, y) |
|  | return np.concatenate((dJdW1.ravel(), dJdW2.ravel())) |
|  |  |
|  |  |
|  | NN = neuralNetwork() |
|  | yHat = NN.forward(X) |
|  |  |
|  | ######################################################################## |
|  | #Bar graph to show error difference prior to training |
|  | ######################################################################## |
|  |  |
|  | bar1 = range(50) |
|  |  |
|  | bar2 = np.arange(0.35,50.35,1) |
|  |  |
|  | plt.bar(bar1, y, width = 0.35, color = "blue", alpha=0.8) |
|  |  |
|  | plt.bar(bar2, yHat, width = 0.35, color = "red", alpha=0.8) |
|  |  |
|  | plt.show() |
|  | ############################################################################ |
|  | #We start to train our network below in a "too basic" gradient descent algorithm |
|  | ############################################################################ |
|  |  |
|  | grad = NN.computeGradients(X,y) |
|  |  |
|  | learningRate = 0.01 #Arbitrarily chosen learning rate for now. Will choose optimal value post reading theory |
|  | #This is almost certainly too high however. |
|  |  |
|  | for i in range(40): #40 is another randomly chosen value. Next step is to stop when our gradients approach zero. |
|  | #Yes that could put us in a local minimum but let's take this one step at a time. |
|  | gradients = NN.costFunctionPrime(X,y) |
|  | dLdW1 = gradients[0] |
|  | dLdW2 = gradients[1] |
|  | NN.W1 -= learningRate\*dLdW1 |
|  | NN.W2 -= learningRate\*dLdW2 |
|  | print(NN.costFunction(X,y)) #In almost all cases our cost function decreases before we "bounce" out of the lowest point. |
|  | #This means our learning rate is too high |
|  |  |
|  |  |
|  |  |

* + © 2019 GitHub, Inc.
  + [Terms](https://github.com/site/terms)
  + [Privacy](https://github.com/site/privacy)
  + [Security](https://github.com/security)
  + [Status](https://githubstatus.com/)
  + [Help](https://help.github.com)
  + [Contact GitHub](https://github.com/contact)
  + [Pricing](https://github.com/pricing)
  + [API](https://developer.github.com)
  + [Training](https://training.github.com)
  + [Blog](https://github.blog)
  + [About](https://github.com/about)