Implementation of ANN in Python

Dhruv Bhargava[#], Rishi Banerjee[*]

[#]*SCIT Department, Manipal University Jaipur*
Jaipur, Rajasthan

[1]`dhruv.muj@gmail.com`

[2]`rishieric91@gmail.com`

***Abstract*— This article is an implementation of a simple ANN in python and demonstrates how powerful it is.**

***Keywords*— ANN, Python, Hidden Layers, Cost, Mean Squared Error**

I. INTRODUCTION

This document demonstrates our implementation of Ann in python using numpy library and its applications in the real world

II. MODEL ARCHITECTURE

Our model consists of

- Two input parameters (which can be changed as per requirement)
- Two hidden layers with 128 neurons and a bias unit in each layer ,each neuron has sigmoid activations
- Output layer consists of 1 neuron which has a sigmoid activation
- The cost function used is mean squared error
- The network is optimized using gradient descent

III. DATASET AND TRAINING

- We generated a low variance dummy data set in order to test the capabilities of this network.
- *We trained the network for 50 epochs with this dataset , using gradient descent and backpropagation*
- Loss Function Used is Mean squared error.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(f_i - y_i)^2$$

where N is the number of data points, f_i the value returned by the model and y_i the actual value for data point i.

IV. CODE

ann.py

```python
import numpy as np
import matplotlib.pyplot as plt



class Neural_Network(object):
    def __init__(self):
        # Define Hyperparameters
        self.inputLayerSize = 2
        self.outputLayerSize = 1
        self.hiddenLayerSize = 128

        # Weights (parameters)
        self.W1 = np.random.randn(self.inputLayerSize, self.hiddenLayerSize)
        self.W2 = np.random.randn(self.hiddenLayerSize, self.hiddenLayerSize)
        self.W3 = np.random.randn(self.hiddenLayerSize, self.outputLayerSize)

        self.b1 = np.random.randn(1, self.hiddenLayerSize)
        self.b2 = np.random.randn(1, self.hiddenLayerSize)
        self.b3 = np.random.randn(1, self.outputLayerSize)

    def forward(self, X):
        # Propogate inputs though network
        self.z2 = np.dot(X, self.W1) + self.b1
        self.a2 = self.sigmoid(self.z2)
        self.z3 = np.dot(self.a2, self.W2) + self.b2
        self.a3 = self.sigmoid(self.z3)
        self.z4 = np.dot(self.a3, self.W3) + self.b3
        yHat = self.z4
        return yHat

    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 costFunction(self, X, y):
        # Compute cost for given X,y, use weights already stored in class.
        self.yHat = self.forward(X)
        J = 0.5 * sum((y - self.yHat) ** 2)
        return J[0]

    def costFunctionPrime(self, X, y):
        # Compute derivative with respect to W and W2 for a given X and y:
        self.yHat = self.forward(X)

        # don't get sigmoid prime in output layer.
        # we don't use activation function in output layer
        delta4 = -(y - self.yHat)
        dJdW3 = np.dot(self.a3.T, delta4)
        dJdb3 = np.mean(delta4, axis=0)

        # after then, we need sigmoid prime because of activate function(sigmoid)
        delta3 = np.dot(delta4, self.W3.T) * self.sigmoidPrime(self.z3)
        dJdW2 = np.dot(self.a2.T, delta3)
        dJdb2 = np.mean(delta3, axis=0)

        delta2 = np.dot(delta3, self.W2.T) * self.sigmoidPrime(self.z2)
        dJdW1 = np.dot(X.T, delta2)
        dJdb1 = np.mean(delta2, axis=0)

        return dJdW1, dJdW2, dJdW3, dJdb1, dJdb2, dJdb3

    def gradient_descent(self, lr, dJdW1, dJdW2, dJdW3, dJdb1, dJdb2, dJdb3):

        self.W1 = self.W1 - lr * dJdW1
        self.W2 = self.W2 - lr * dJdW2
        self.W3 = self.W3 - lr * dJdW3

        self.b1 = self.b1 - lr * dJdb1
        self.b2 = self.b2 - lr * dJdb2
        self.b3 = self.b3 - lr * dJdb3
```

V. RESULTS

The network converges well with-in the 50 epochs and performs reasonably well

This model can however be fit to even more complex data sets and still manage to achieve reasonable accuracy over them