Computational Neuro- Assignment 1- Parts A-B

Ayala Bouhnik-Gelbord
Jonatan Boritsky
Ginton Durlacher
Gil Tzioni

Computational neuro- Assignment 1-

In this assignment we were requested to implement Adaline algorithm and Conclusions about the algorithm.

Part A-

Data set-

x, y <= 100. The data is all data points where x is of the form m/100 where m is an integer between -10000 and +10000 and y is of the form n/100 with n an integer between - 10000 and +10000. All data points with y >1 have the value 1, all other points have the value -1.
We are given a random sample of data of size 1000 together with its value (e.g. the point <601/100, 802/100> has value 1; while the point <8000/100, 70/100> has the value -1).

After we created the dataset and implemented the algorithm we ran our model a few times. (each time we changed something else, the test set, the alpha (learning rate) or the number of iteration).
The results we got were amazing and very high, and a change in the values of the parameters did not lead to a significant change in the percentages of accuracy.

Here we can see that the influence of the alpha and the test set on the accuracy rate:

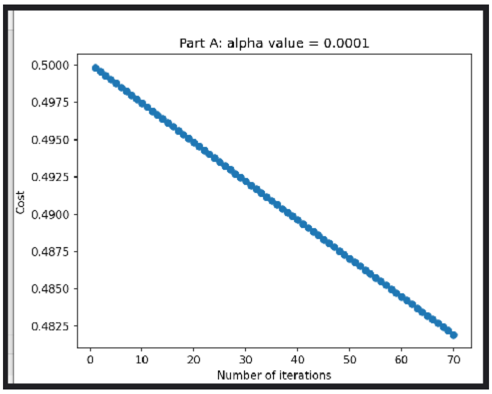
First test set-

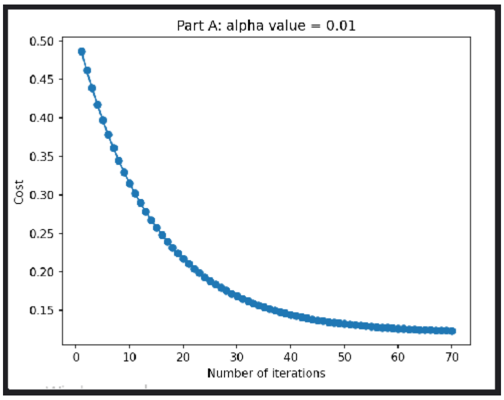
```
-------------Part A-------------

alpha: 0.0001 , data = 1,000 , n = 10000
Accuracy percentages:  95.8 %
cost:  0.48198532389173915

alpha: 0.01 , data = 1,000 , n = 10000
Accuracy percentages:  99.5 %
cost:  0.12396601454251624

alpha: 0.1 , data = 1,000 , n = 10000
Accuracy percentages:  99.5 %
cost:  0.12113889000942514
```

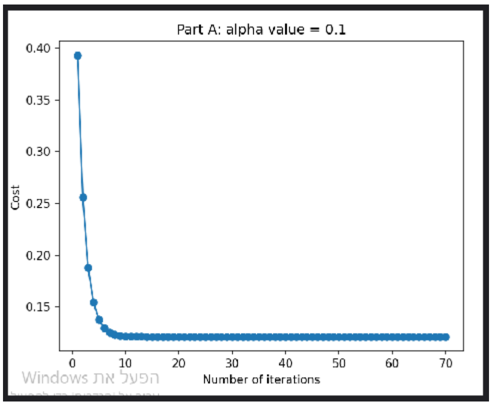
Second test set-

```
-------------Part A-------------

alpha: 0.0001 , data = 1,000 , n = 10000
Accuracy percentages:  93.7 %
cost:  0.4832995386162738

alpha: 0.01 , data = 1,000 , n = 10000
Accuracy percentages:  99.2 %
cost:  0.13088437918087845

alpha: 0.1 , data = 1,000 , n = 10000
Accuracy percentages:  99.5 %
cost:  0.1267048067825935
```

Here we can see that all the alpha values between 0.01 to 1 will bring great accuracy, and that the Adaline brings a great result in both test cases.







Here you can see the ratio between the number of iterations to the cost value (in each table the alpha rate is different) , the cost remains low.

The result of the algorithm on the data set is so high because our data is linear and Adaline has a single layer so it can perform well on linear data.

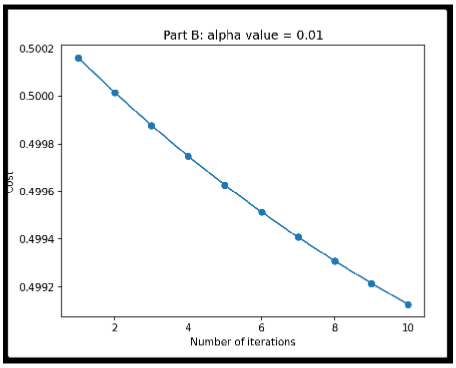
Part B-

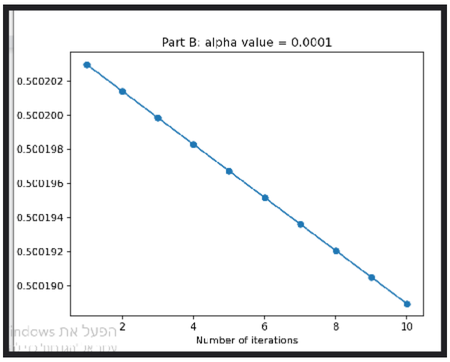
Data Set-
same data as part A but now points such that <x.y> has value 1 only if $4 <= x^2 + y^2 <= 9$ (A canonical circle equation).

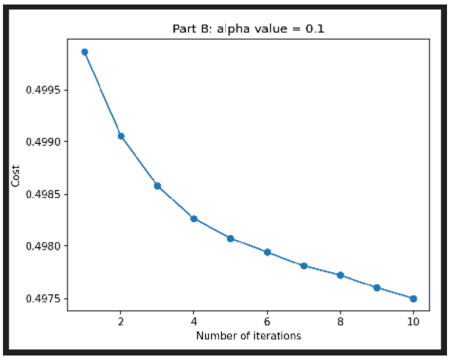
After we created the required data set we ran Adaline algorithm on the data set. We got very poor results.
We tried to change the test set, the alpha (learning rate) or the number of iteration but the accuracy percentages remained low.

```
-------------Part B-------------

alpha: 0.0001 , data = 1,000 , n = 10000
Accuracy percentages:  46.400000000000006 %
cost:  0.5001889386989082

alpha: 0.01 , data = 1,000 , n = 10000
Accuracy percentages:  52.300000000000004 %
cost:  0.4991268365186632

alpha: 0.1 , data = 1,000 , n = 10000
Accuracy percentages:  51.6 %
cost:  0.4975006874552314
```

Here we can see that all the alpha values give us very poor results.







Here you can see the ratio between the number of iterations to the cost value (in each table the alpha rate is different), the cost remains high.

The reason we got low accuracy is because this data is non-linear and an Adaline algorithm is not good for non-linear data sets (because it has a single layer). If we want to get better results on this data we need to use a neural network with Multiple layers.

Another issue is that the range $4 <= x^2 + y^2 <= 9$ is very small and the vast majority of the points will be out of the circle.
If we increase the range we are likely to get larger accuracy percentages, but still not very good on

Code:

Adaline implementation:

```
import numpy as np


class Adaline_imp:
   def __init__(self, alpha=0.01, number_of_iterations=100, shuff=True):
       self.number_of_iterations = number_of_iterations
       self.alpha = alpha
       self.shuff = shuff
       self.weights = []
       self.loss_avg = []

   '''
     From prof. L. Manevitz slides:
     •    1. Apply input to Adaline input
     •    2. Find the square error of current input
       - Errsq(k) = (d(k) -  W x(k))**2
     •    3. Approximate Grad(ErrorSquare) by
       -  differentiating Errsq
       -  approximating average Errsq by Errsq(k)
       -  obtain  -2Error (k)x(k)    Also called "delta" rule  -2deltaX(k)
       4. Update W:   W(new) = W(old) + 2mdX(k)
       5.  Repeat steps 1 to 4.
   '''

   def fit(self, x_test, y_train):
       row = x_test.shape[0]
       col = x_test.shape[1]

       bias = np.ones((row, col + 1))
       bias[:, 1:] = x_test
       x_test = bias

       np.random.seed(1)
       self.weights = np.random.rand(col + 1)

       for i in range(self.number_of_iterations):
           if self.shuff:
               x_test, y_train = self.shuffle_data(x_test, y_train)
           loss_arr = []
           for xi, target in zip(x_test, y_train):
               loss_arr.append(self.upd_weights(xi, target))
           avg = sum(loss_arr) / len(y_train)
```

```
            self.loss_avg.append(avg)

        return self

    def upd_weights(self, xi, target):
        inputs_cal = self.net_input_calculation((xi/100)) # We divided it by 
100 because we get very high numbers and it is difficult to calculate them. 
NOTE- in the beginning we didn't divide xi and we got runtime warnings and 
we didn't understand why, but in the end we succeeded to fix it.
        error = target - inputs_cal
        self.weights += self.alpha * (xi/100).dot(error) # We divided it by 
100 because we get very high numbers and it is difficult to calculate them.
        cost = 0.5 * (error ** 2)
        return cost

    def shuffle_data(self, x_test, y_train):
        i = np.random.permutation(len(y_train))
        return x_test[i], y_train[i]

    def net_input_calculation(self, x_test):
        ans = ((x_test/100) @ self.weights) # calculates the inputs of the 
neural network
        return ans

    def activation_function(self, x_test):
        return self.net_input_calculation(x_test)

    def predict(self, x_test):
        if type(x_test) is list: x_test = np.array(x_test)
        if len(x_test.T) != len(self.weights):
            bias = np.ones((x_test.shape[0], x_test.shape[1] + 1))
            bias[:, 1:] = x_test
            x_test = bias
        return np.where(self.activation_function(x_test) > 0.0, 1, -1)

    def score(self, x_test, y_train):
        misclassified_data_count = abs((self.predict(x_test) - y_train) / 
2).sum()
        total_data_count = len(x_test)
        self.score_ = (total_data_count - misclassified_data_count) / 
total_data_count
        return self.score_
```

Main:

```
# This is the main file of the project, we create the data and the tables.
import numpy as np
import matplotlib.pyplot as plt
import random
from matplotlib.colors import ListedColormap
from Adaline_imp import Adaline_imp

size = 1000  # data of size 1000


def data_points(n, part):
   data = np.empty((size, 2), dtype=object)
   # random.seed(10)
   # fill the array with random points where x is of the form m/100 where m
is an
   # integer between -10000 and +10000 and y is of the form n/100 with n an
integer between -
   # 10000 and +10000.
   for i in range(size):
       data[i, 0] = ((random.randint(-n, n)) / 100)
       data[i, 1] = ((random.randint(-n, n)) / 100)

   train = np.zeros(size)

   # all data points with y >1 have the value 1; all other points have the
value -1
   if part == "part A":
       for i in range(size):
           if data[i][1] > 1:
               train[i] = 1
           else:
               train[i] = -1

   if part == "part B":
       for i in range(size):
           if 4 <= ((data[i][1] ** 2) + (data[i][0] ** 2)) <= 9:
               train[i] = 1
           else:
               train[i] = -1

   x_test = data.astype(np.float64)
   y_train = train.astype(np.float64)
```

```
    return x_test, y_train

def part_A():
    print("\n____________________Part A____________________\n")
    n = 10000


    x_test, y_train = data_points(n, "part A")



#---------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------

    adaline_A_1 = Adaline_imp(0.0001, 70).fit(x_test, y_train)

    adaline_A_2 = Adaline_imp(0.01, 70).fit(x_test, y_train)

    adaline_A_3 = Adaline_imp(0.1, 70).fit(x_test, y_train)

    A= "Part A"
    print_tables(x_test, y_train, adaline_A_1, A)
    print_tables(x_test, y_train, adaline_A_2, A)
    print_tables(x_test, y_train, adaline_A_3, A)


    print("alpha: 0.0001 , data = 1,000 , n = 10000")
    print("Accuracy percentages: ", adaline_A_1.score(x_test, y_train) * 100,
"%")
    print("cost: ", np.array(adaline_A_1.loss_avg).min(),"\n")


    print("alpha: 0.01 , data = 1,000 , n = 10000")
    print("Accuracy percentages: ", adaline_A_2.score(x_test, y_train) * 100,
"%")
    print("cost: ", np.array(adaline_A_2.loss_avg).min(), "\n")

    print("alpha: 0.1 , data = 1,000 , n = 10000")
    print("Accuracy percentages: ", adaline_A_3.score(x_test, y_train) * 100,
"%")
    print("cost: ", np.array(adaline_A_3.loss_avg).min(), "\n")


def part_B():
    print("\n_____________________Part B____________________   \n")
    x_test, y_train = data_points(10000, "part B")

    adaline_B_1 = Adaline_imp(0.0001, 10).fit(x_test, y_train)

    adaline_B_2 = Adaline_imp(0.01, 10).fit(x_test, y_train)

    adaline_B_3 = Adaline_imp(0.1, 10).fit(x_test, y_train)
```

```python
    B = "Part B"
    print_tables(x_test, y_train, adaline_B_1, B)
    print_tables(x_test, y_train, adaline_B_2, B)
    print_tables(x_test, y_train, adaline_B_3, B)


    print("alpha: 0.0001 , data = 1,000 , n = 10000")
    print("Accuracy percentages: ", adaline_B_1.score(x_test, y_train) * 100,
"%")
    print("cost: ", np.array(adaline_B_1.loss_avg).min() , "\n")

    print("alpha: 0.01 , data = 1,000 , n = 10000")
    print("Accuracy percentages: ", adaline_B_2.score(x_test, y_train) * 100,
"%")
    print("cost: ", np.array(adaline_B_2.loss_avg).min(), "\n")

    print("alpha: 0.1 , data = 1,000 , n = 10000")
    print("Accuracy percentages: ", adaline_B_3.score(x_test, y_train) * 100,
"%")
    print("cost: ", np.array(adaline_B_3.loss_avg).min(), "\n")

#----------------------------------------------------------------------------------------------------------------------------------------------------------------------------
#
# ------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

# print table's

def print_tables(adaline, part):
    fig, ax = plt.subplots()
    ax.plot(range(1, len(adaline.loss_avg) + 1), adaline.loss_avg, marker='o')
    ax.set_xlabel('Number of iterations')
    ax.set_ylabel('Cost')
    plt.title(part + ": alpha value = " + adaline.alpha.__str__())

    plt.show()



if __name__== '__main__':
    part_A()
    part_B()
```