Laboratory 2 Assignment

Galer Gabriel 442G

# Laboratory 1 – Multiclass SVM

## Exercise 6 and 7

import numpy as np

def predict(xsample, W):

    s = []

    s = W.dot(xsample)

    return s

# TODO - Application 3 - Step 3 - The function that compute the loss for a data point

def computeLossForASample(s, labelForSample, delta):

    loss\_i = 0

    syi = s[

        labelForSample

    ]  # the score for the correct class corresonding to the current input sample based on the label yi

    for idx, sj in enumerate(s):

        if idx != labelForSample:

            loss\_i += max(0, sj - syi + delta)

    return loss\_i

# TODO - Application 3 - Step 4 - The function that compute the gradient loss for a data point

def computeLossGradientForASample(W, s, currentDataPoint, labelForSample, delta):

    dW\_i = np.zeros(W.shape)  # initialize the matrix of gradients with zero

    syi = s[labelForSample]  # establish the score obtained for the true class

    for j, sj in enumerate(s):

        dist = sj - syi + delta

        if j == labelForSample:

            continue

        if dist > 0:

            dW\_i[j] = currentDataPoint

            dW\_i[labelForSample] = dW\_i[labelForSample] - currentDataPoint

    return dW\_i

def main():

    # Input points in the 4 dimensional space

    x\_train = np.array(

        [

            [1, 5, 1, 4],

            [2, 4, 0, 3],

            [2, 1, 3, 3],

            [2, 0, 4, 2],

            [5, 1, 0, 2],

            [4, 2, 1, 1],

        ]

    )

    # Labels associated with the input points

    y\_train = [0, 0, 1, 1, 2, 2]

    # Input points for prediction

    x\_test = np.array([[1, 5, 2, 4], [2, 1, 2, 3], [4, 1, 0, 1]])

    # Labels associated with the testing points

    y\_test = [0, 1, 2]

    # The matrix of wights

    W = np.array([[-1, 2, 1, 3], [2, 0, -1, 4], [1, 3, 2, 1]])

    delta = 1  # margin

    step\_size = 0.01  # weights adjustment ratio

    loss\_L = 0

    dW = np.zeros(W.shape)

    prev\_loss = 99

    epochs = 0

    while abs(loss\_L - prev\_loss) > 0.001:

        epochs += 1

        prev\_loss = loss\_L

        # TODO - Application 3 - Step 2 - For each input data...

        for idx, xsample in enumerate(x\_train):

            print(xsample)

            # TODO - Application 3 - Step 2 - ...compute the scores s for all classes (call the method predict)

            s = predict(xsample, W)

            # TODO - Application 3 - Step 3 - Call the function (computeLossForASample) that

            #  compute the loss for a data point (loss\_i)

            loss\_i = computeLossForASample(s, y\_train[idx], delta)

            # Print the scores - Uncomment this

            # print(

            #     "Scores for sample {} with label {} is: {} and loss is {}".format(

            #         idx, y\_train[idx], s, loss\_i

            #     )

            # )

            # TODO - Application 3 - Step 4 - Call the function (computeLossGradientForASample) that

            #  compute the gradient loss for a data point (dW\_i)

            dW\_i = computeLossGradientForASample(W, s, x\_train[idx], y\_train[idx], delta)

            # TODO - Application 3 - Step 5 - Compute the global loss for all the samples (loss\_L)

            loss\_L += loss\_i

            # TODO - Application 3 - Step 6 - Compute the global gradient loss matrix (dW)

            dW += dW\_i

        # TODO - Application 3 - Step 7 - Compute the global normalized loss

        loss\_L\_norm = loss\_L / len(y\_train)

        # print("The global normalized loss = {}".format(loss\_L\_norm))

        # TODO - Application 3 - Step 8 - Compute the global normalized gradient loss matrix

        dW = dW / len(y\_train)

        # TODO - Application 3 - Step 9 - Adjust the weights matrix

        W = W - step\_size \* dW

        print(abs(prev\_loss - loss\_L))

    print(f"Number of epochs: {epochs}")

    # TODO - Exercise 7 - After solving exercise 6, predict the labels for the points existent in x\_test variable

    #  and compare them with the ground truth labels. What is the system accuracy?

    correctPredicted = 0

    for idx, xsample in enumerate(x\_test):

        if np.argmax(predict(xsample, W)) == y\_test[idx]:

            correctPredicted += 1

    accuracy = correctPredicted / len(y\_test) \* 100

    print("Accuracy for test = {}%".format(accuracy))

    return

if \_\_name\_\_ == "\_\_main\_\_":

    main()

## Exercise 8

import pandas as pd

import numpy as np

import random

def get\_iris\_data(path):

    df = pd.read\_csv(path)

    headers = [

        'SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm', 'Class'

    ]

    x = []

    y = []

    for row in [x[1] for x in df.iterrows()]:

        xs = []

        for id\_h, h in enumerate(headers):

            if id\_h == len(headers) - 1:

                y.append(row[h])

                x.append(xs)

            else:

                xs.append(row[h])

    d = {}

    cnt = 0

    for label in y:

        if label not in d:

            d[label] = cnt

            cnt += 1

    for id, label in enumerate(y):

        y[id] = d[label]

    print(y)

    return x, y

def predict(xsample, W):

    s = []

    s = W.dot(xsample)

    return s

# TODO - Application 3 - Step 3 - The function that compute the loss for a data point

def computeLossForASample(s, labelForSample, delta):

    loss\_i = 0

    syi = s[

        labelForSample

    ]  # the score for the correct class corresonding to the current input sample based on the label yi

    for idx, sj in enumerate(s):

        if idx != labelForSample:

            loss\_i += max(0, sj - syi + delta)

    return loss\_i

# TODO - Application 3 - Step 4 - The function that compute the gradient loss for a data point

def computeLossGradientForASample(W, s, currentDataPoint, labelForSample, delta):

    dW\_i = np.zeros(W.shape)  # initialize the matrix of gradients with zero

    syi = s[labelForSample]  # establish the score obtained for the true class

    for j, sj in enumerate(s):

        dist = sj - syi + delta

        if j == labelForSample:

            continue

        if dist > 0:

            dW\_i[j] = currentDataPoint

            dW\_i[labelForSample] = dW\_i[labelForSample] - currentDataPoint

    return dW\_i

def main():

    # Input points in the 4 dimensional space

    # x\_train = np.array(

    #     [

    #         [1, 5, 1, 4],

    #         [2, 4, 0, 3],

    #         [2, 1, 3, 3],

    #         [2, 0, 4, 2],

    #         [5, 1, 0, 2],

    #         [4, 2, 1, 1],

    #     ]

    # )

    # # Labels associated with the input points

    # y\_train = [0, 0, 1, 1, 2, 2]

    # # Input points for prediction

    # x\_test = np.array([[1, 5, 2, 4], [2, 1, 2, 3], [4, 1, 0, 1]])

    # # Labels associated with the testing points

    # y\_test = [0, 1, 2]

    x, y = map(np.array, get\_iris\_data('./Iris.csv'))

    random.shuffle(x)

    random.shuffle(y)

    N = len(x)

    train\_limit = int(N \* 0.2)

    x\_train = x[:train\_limit]

    x\_test = x[train\_limit:]

    y\_train = y[:train\_limit]

    y\_test = y[train\_limit:]

    # print(x\_train, x\_test, y\_train, y\_test)

    # The matrix of wights

    # W = np.array([[-1, 2, 1, 3], [2, 0, -1, 4], [1, 3, 2, 1]], dtype=np.float)

    # W = np.array(np.random.uniform(size=(150, 4)), dtype=np.float)

    W = np.zeros((150, 4), dtype=np.float)

    delta = 0.1  # margin

    step\_size = 1  # weights adjustment ratio

    loss\_L = 0

    dW = np.zeros(W.shape)

    epochs = 0

    accuracy = 0

    while accuracy <= 90:

        epochs += 1

        # TODO - Application 3 - Step 2 - For each input data...

        for idx, xsample in enumerate(x\_train):

            # TODO - Application 3 - Step 2 - ...compute the scores s for all classes (call the method predict)

            s = predict(xsample, W)

            # TODO - Application 3 - Step 3 - Call the function (computeLossForASample) that

            #  compute the loss for a data point (loss\_i)

            loss\_i = computeLossForASample(s, y\_train[idx], delta)

            # Print the scores - Uncomment this

            # print(

            #     "Scores for sample {} with label {} is: {} and loss is {}".format(

            #         idx, y\_train[idx], s, loss\_i

            #     )

            # )

            # TODO - Application 3 - Step 4 - Call the function (computeLossGradientForASample) that

            #  compute the gradient loss for a data point (dW\_i)

            dW\_i = computeLossGradientForASample(W, s, x\_train[idx], y\_train[idx], delta)

            # TODO - Application 3 - Step 5 - Compute the global loss for all the samples (loss\_L)

            loss\_L += loss\_i

            # TODO - Application 3 - Step 6 - Compute the global gradient loss matrix (dW)

            dW += dW\_i

        # TODO - Application 3 - Step 7 - Compute the global normalized loss

        loss\_L\_norm = loss\_L / len(y\_train)

        # print("The global normalized loss = {}".format(loss\_L\_norm))

        # TODO - Application 3 - Step 8 - Compute the global normalized gradient loss matrix

        dW = dW / len(y\_train)

        # TODO - Application 3 - Step 9 - Adjust the weights matrix

        W = W - step\_size \* dW

        correctPredicted = 0

        for idx, xsample in enumerate(x\_test):

            if np.argmax(predict(xsample, W)) == y\_test[idx]:

                correctPredicted += 1

        accuracy = correctPredicted / len(y\_test) \* 100

        print("Accuracy for test = {}%".format(accuracy))

    return

if \_\_name\_\_ == "\_\_main\_\_":

    main()

# Laboratory 2 – XOR and AND regression

## Exercise 1

import numpy as np

def activationFunction(n):

    # TODO - Application 1 - Step 4b - Define the binary step function as activation function

    return 1 if n >= 0 else 0

def forwardPropagation(p, weights, bias):

    a = None  # the neuron output

    # TODO - Application 1 - Step 4a - Multiply weights with the input vector (p) and add the bias   =>  n

    res = np.dot(p, weights) + bias

    # TODO - Application 1 - Step 4c - Pass the result to the activation function  =>  a

    a = activationFunction(res)

    return a

def main():

    # Application 1 - Train a single neuron perceptron in order to predict the output of an AND gate.

    # The network should receive as input two values (0 or 1) and should predict the target output

    # Input data

    P = [[0, 0], [0, 1], [1, 0], [1, 1]]

    # Labels

    t = [0, 0, 0, 1]

    # TODO - Application 1 - Step 2 - Initialize the weights with zero  (weights)

    w = [0, 0]

    # TODO - Application 1 - Step 2 - Initialize the bias with zero  (bias)

    b = 0

    # TODO - Application 1 - Step 3 - Set the number of training steps  (epochs)

    epochs = 0

    # TODO - Application 1 - Step 4 - Perform the neuron training for multiple epochs

    while w != [2, 1] or b != -3:

        epochs += 1

        for i in range(len(t)):

            # TODO - Application 1 - Step 4 - Call the forwardPropagation method

            prediction = forwardPropagation(P[i], w, b)

            # TODO - Application 5 - Compute the prediction error (error)

            err = t[i] - prediction

            # TODO - Application 6 - Update the weights

            w[0] += err \* P[i][0]

            w[1] += err \* P[i][1]

            # should work?

            # w += err \* P

            # TODO - Update the bias

            b += err

    # TODO - Application 1 - Print weights and bias

    print(f"weights: {w}, bias: {b}")

    print(f"epochs needed: {epochs}")

    # TODO - Application 1 - Step 7 - Display the results

    return

if \_\_name\_\_ == "\_\_main\_\_":

    main()

## Exercise 2

import numpy as np

def activationFunction(n):

    # TODO - Application 1 - Step 4b - Define the binary step function as activation function

    return 1 if n >= 0 else 0

def forwardPropagation(p, weights, bias):

    a = None  # the neuron output

    # TODO - Application 1 - Step 4a - Multiply weights with the input vector (p) and add the bias   =>  n

    res = np.dot(p, weights) + bias

    # TODO - Application 1 - Step 4c - Pass the result to the activation function  =>  a

    a = activationFunction(res)

    return a

def main():

    # Application 1 - Train a single neuron perceptron in order to predict the output of an AND gate.

    # The network should receive as input two values (0 or 1) and should predict the target output

    # Input data

    P = [[0, 0], [0, 1], [1, 0], [1, 1]]

    # Labels

    t = [0, 1, 1, 1]

    # TODO - Application 1 - Step 2 - Initialize the weights with zero  (weights)

    w = [0, 0]

    # TODO - Application 1 - Step 2 - Initialize the bias with zero  (bias)

    b = 0

    # TODO - Application 1 - Step 3 - Set the number of training steps  (epochs)

    epochs = 100

    # TODO - Application 1 - Step 4 - Perform the neuron training for multiple epochs

    for ep in range(epochs):

        for i in range(len(t)):

            # TODO - Application 1 - Step 4 - Call the forwardPropagation method

            prediction = forwardPropagation(P[i], w, b)

            # TODO - Application 5 - Compute the prediction error (error)

            err = t[i] - prediction

            # TODO - Application 6 - Update the weights

            w[0] += err \* P[i][0]

            w[1] += err \* P[i][1]

            # should work?

            # w += err \* P

            # TODO - Update the bias

            b += err

    # TODO - Application 1 - Print weights and bias

    print(f"weights: {w}, bias: {b}")

    # TODO - Application 1 - Step 7 - Display the results

    return

if \_\_name\_\_ == "\_\_main\_\_":

    main()

## Exercise 3

import numpy as np

def activationFunction(n):

    # TODO - Application 1 - Step 4b - Define the binary step function as activation function

    return 1.0 / (1.0 + np.exp(-n))

def forwardPropagation(p, weights, bias):

    a = None  # the neuron output

    # TODO - Application 1 - Step 4a - Multiply weights with the input vector (p) and add the bias   =>  n

    res = np.dot(p, weights) + bias

    # TODO - Application 1 - Step 4c - Pass the result to the activation function  =>  a

    a = activationFunction(res)

    return a

def main():

    # Application 1 - Train a single neuron perceptron in order to predict the output of an AND gate.

    # The network should receive as input two values (0 or 1) and should predict the target output

    # Input data

    P = [[0, 0], [0, 1], [1, 0], [1, 1]]

    # Labels

    t = [0, 0, 0, 1]

    # TODO - Application 1 - Step 2 - Initialize the weights with zero  (weights)

    w = [0, 0]

    # TODO - Application 1 - Step 2 - Initialize the bias with zero  (bias)

    b = 0

    # TODO - Application 1 - Step 3 - Set the number of training steps  (epochs)

    epochs = 100

    # TODO - Application 1 - Step 4 - Perform the neuron training for multiple epochs

    for ep in range(epochs):

        for i in range(len(t)):

            # TODO - Application 1 - Step 4 - Call the forwardPropagation method

            prediction = forwardPropagation(P[i], w, b)

            # TODO - Application 5 - Compute the prediction error (error)

            err = t[i] - prediction

            # TODO - Application 6 - Update the weights

            w[0] += err \* P[i][0]

            w[1] += err \* P[i][1]

            # should work?

            # w += err \* P

            # TODO - Update the bias

            b += err

    # TODO - Application 1 - Print weights and bias

    print(f"weights: {w}, bias: {b}")

    # TODO - Application 1 - Step 7 - Display the results

    return

if \_\_name\_\_ == "\_\_main\_\_":

    main()

## Exercise 4

import numpy as np

def sigmoid(n):

    return 1.0 / (1.0 + np.exp(-n))

def sigmoidDerivative(n):

    return n \* (1 - n)

def error\_function(output, prediction):

    s = 0

    for i in range(len(output)):

        s += (output[i] - prediction[i]) \*\* 2

    s /= 2 \* len(output)

    return s

def forwardPropagationLayer(p, weights, biases):

    a = None  # the layer output

    # Multiply weights with the input vector (p) and add the bias   =>  n

    n = np.dot(p, weights) + biases

    # Pass the result to the activation function  =>  a

    a = sigmoid(n)

    return a

def main():

    # Application 2 - Train a ANN in order to predict the output of an XOR gate.

    # The network should receive as input two values (0 or 1) and should predict the target output

    # Input data

    points = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

    # Labels

    labels = np.array([[0], [1], [1], [0]])

    # Initialize the weights and biases with random values

    inputSize = 2

    noNeuronsLayer1 = 2

    noNeuronsLayer2 = 1

    weightsLayer1 = np.random.uniform(size=(inputSize, noNeuronsLayer1))

    weightsLayer2 = np.random.uniform(size=(noNeuronsLayer1, noNeuronsLayer2))

    biasLayer1 = np.random.uniform(size=(1, noNeuronsLayer1))

    biasLayer2 = np.random.uniform(size=(1, noNeuronsLayer2))

    noEpochs = 0

    learningRate = 0.3

    total\_error = 999

    # Train the network for noEpochs

    while total\_error >= 0.01:

        noEpochs += 1

        total\_error = 0

        # Forward Propagation

        hidden\_layer\_output = forwardPropagationLayer(points, weightsLayer1, biasLayer1)

        predicted\_output = forwardPropagationLayer(

            hidden\_layer\_output, weightsLayer2, biasLayer2

        )

        # Backpropagation

        bkProp\_error = labels - predicted\_output

        d\_predicted\_output = bkProp\_error \* sigmoidDerivative(predicted\_output)

        error\_hidden\_layer = d\_predicted\_output.dot(weightsLayer2.T)

        d\_hidden\_layer = error\_hidden\_layer \* sigmoidDerivative(hidden\_layer\_output)

        # Updating Weights and Biases

        weightsLayer2 = (

            weightsLayer2 + hidden\_layer\_output.T.dot(d\_predicted\_output) \* learningRate

        )

        biasLayer2 = (

            biasLayer2

            + np.sum(d\_predicted\_output, axis=0, keepdims=True) \* learningRate

        )

        weightsLayer1 = weightsLayer1 + points.T.dot(d\_hidden\_layer) \* learningRate

        biasLayer1 = (

            biasLayer1 + np.sum(d\_hidden\_layer, axis=0, keepdims=True) \* learningRate

        )

        total\_error = error\_function(labels, predicted\_output)

        print(noEpochs)

    print(f"Needed epochs: {noEpochs}")

    # Print weights and bias

    print("weightsLayer1 = {}".format(weightsLayer1))

    print("biasesLayer1 = {}".format(biasLayer1))

    print("weightsLayer2 = {}".format(weightsLayer2))

    print("biasLayer2 = {}".format(biasLayer2))

    # Display the results

    for i in range(len(labels)):

        outL1 = forwardPropagationLayer(points[i], weightsLayer1, biasLayer1)

        outL2 = forwardPropagationLayer(outL1, weightsLayer2, biasLayer2)

        print(

            "Input = {} - Predict = {} - Label = {}".format(points[i], outL2, labels[i])

        )

if \_\_name\_\_ == "\_\_main\_\_":

    main()

## Exercise 5

import numpy as np

def sigmoid(n):

    return 2.0 / (1.0 + np.exp(-2 \* n)) - 1

def sigmoidDerivative(n):

    return (1 + n) \* (1 - n)

def forwardPropagationLayer(p, weights, biases):

    a = None  # the layer output

    # Multiply weights with the input vector (p) and add the bias   =>  n

    n = np.dot(p, weights) + biases

    # Pass the result to the activation function  =>  a

    a = sigmoid(n)

    return a

def main():

    # Application 2 - Train a ANN in order to predict the output of an XOR gate.

    # The network should receive as input two values (0 or 1) and should predict the target output

    # Input data

    points = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

    # Labels

    labels = np.array([[0], [1], [1], [0]])

    # Initialize the weights and biases with random values

    inputSize = 2

    noNeuronsLayer1 = 2

    noNeuronsLayer2 = 1

    weightsLayer1 = np.random.uniform(size=(inputSize, noNeuronsLayer1))

    weightsLayer2 = np.random.uniform(size=(noNeuronsLayer1, noNeuronsLayer2))

    biasLayer1 = np.random.uniform(size=(1, noNeuronsLayer1))

    biasLayer2 = np.random.uniform(size=(1, noNeuronsLayer2))

    noEpochs = 5000

    learningRate = 0.3

    # Train the network for noEpochs

    for \_ in range(noEpochs):

        # Forward Propagation

        hidden\_layer\_output = forwardPropagationLayer(points, weightsLayer1, biasLayer1)

        predicted\_output = forwardPropagationLayer(

            hidden\_layer\_output, weightsLayer2, biasLayer2

        )

        # Backpropagation

        bkProp\_error = labels - predicted\_output

        d\_predicted\_output = bkProp\_error \* sigmoidDerivative(predicted\_output)

        error\_hidden\_layer = d\_predicted\_output.dot(weightsLayer2.T)

        d\_hidden\_layer = error\_hidden\_layer \* sigmoidDerivative(hidden\_layer\_output)

        # Updating Weights and Biases

        weightsLayer2 = (

            weightsLayer2 + hidden\_layer\_output.T.dot(d\_predicted\_output) \* learningRate

        )

        biasLayer2 = (

            biasLayer2

            + np.sum(d\_predicted\_output, axis=0, keepdims=True) \* learningRate

        )

        weightsLayer1 = weightsLayer1 + points.T.dot(d\_hidden\_layer) \* learningRate

        biasLayer1 = (

            biasLayer1 + np.sum(d\_hidden\_layer, axis=0, keepdims=True) \* learningRate

        )

    # Print weights and bias

    print("weightsLayer1 = {}".format(weightsLayer1))

    print("biasesLayer1 = {}".format(biasLayer1))

    print("weightsLayer2 = {}".format(weightsLayer2))

    print("biasLayer2 = {}".format(biasLayer2))

    # Display the results

    for i in range(len(labels)):

        outL1 = forwardPropagationLayer(points[i], weightsLayer1, biasLayer1)

        outL2 = forwardPropagationLayer(outL1, weightsLayer2, biasLayer2)

        print(

            "Input = {} - Predict = {} - Label = {}".format(points[i], outL2, labels[i])

        )

if \_\_name\_\_ == "\_\_main\_\_":

    main()

## Exercise 6

import numpy as np

def sigmoid(n):

    return 1.0 / (1.0 + np.exp(-n))

def sigmoidDerivative(n):

    return n \* (1 - n)

def forwardPropagationLayer(p, weights, biases):

    a = None  # the layer output

    # Multiply weights with the input vector (p) and add the bias   =>  n

    n = np.dot(p, weights) + biases

    # Pass the result to the activation function  =>  a

    a = sigmoid(n)

    return a

def main():

    # Application 2 - Train a ANN in order to predict the output of an XOR gate.

    # The network should receive as input two values (0 or 1) and should predict the target output

    # Input data

    points = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

    # Labels

    labels = np.array([[0], [1], [1], [0]])

    # Initialize the weights and biases with random values

    inputSize = 2

    noNeuronsLayer1 = 2

    noNeuronsLayer2 = 1

    weightsLayer1 = np.random.standard\_normal(size=(inputSize, noNeuronsLayer1))

    weightsLayer2 = np.random.standard\_normal(size=(noNeuronsLayer1, noNeuronsLayer2))

    biasLayer1 = np.random.standard\_normal(size=(1, noNeuronsLayer1))

    biasLayer2 = np.random.standard\_normal(size=(1, noNeuronsLayer2))

    noEpochs = 5000

    learningRate = 0.3

    # Train the network for noEpochs

    for \_ in range(noEpochs):

        # Forward Propagation

        hidden\_layer\_output = forwardPropagationLayer(points, weightsLayer1, biasLayer1)

        predicted\_output = forwardPropagationLayer(

            hidden\_layer\_output, weightsLayer2, biasLayer2

        )

        # Backpropagation

        bkProp\_error = labels - predicted\_output

        d\_predicted\_output = bkProp\_error \* sigmoidDerivative(predicted\_output)

        error\_hidden\_layer = d\_predicted\_output.dot(weightsLayer2.T)

        d\_hidden\_layer = error\_hidden\_layer \* sigmoidDerivative(hidden\_layer\_output)

        # Updating Weights and Biases

        weightsLayer2 = (

            weightsLayer2 + hidden\_layer\_output.T.dot(d\_predicted\_output) \* learningRate

        )

        biasLayer2 = (

            biasLayer2

            + np.sum(d\_predicted\_output, axis=0, keepdims=True) \* learningRate

        )

        weightsLayer1 = weightsLayer1 + points.T.dot(d\_hidden\_layer) \* learningRate

        biasLayer1 = (

            biasLayer1 + np.sum(d\_hidden\_layer, axis=0, keepdims=True) \* learningRate

        )

    # Print weights and bias

    print("weightsLayer1 = {}".format(weightsLayer1))

    print("biasesLayer1 = {}".format(biasLayer1))

    print("weightsLayer2 = {}".format(weightsLayer2))

    print("biasLayer2 = {}".format(biasLayer2))

    # Display the results

    for i in range(len(labels)):

        outL1 = forwardPropagationLayer(points[i], weightsLayer1, biasLayer1)

        outL2 = forwardPropagationLayer(outL1, weightsLayer2, biasLayer2)

        print(

            "Input = {} - Predict = {} - Label = {}".format(points[i], outL2, labels[i])

        )

if \_\_name\_\_ == "\_\_main\_\_":

    main()

# Observations

## L1E6-7

159 epochs with 100% accuracy

## L2E1

weights: [2, 1], bias: -3

epochs needed: 5

## L2E2

weights: [1, 1], bias: -1

## L2E3

weights: [5.92841512703933, 5.8711073021902], bias: -8.922822364626157

## L2E4

Needed epochs: 2095

weightsLayer1 = [[5.33327418 2.77105156]

[5.18444757 2.75002998]]

biasesLayer1 = [[-1.85605847 -4.13562765]]

weightsLayer2 = [[ 5.91313921]

[-6.13546982]]

biasLayer2 = [[-2.64684549]]

Input = [0 0] - Predict = [[0.12518883]] - Label = [0]

Input = [0 1] - Predict = [[0.86222114]] - Label = [1]

Input = [1 0] - Predict = [[0.86301237]] - Label = [1]

Input = [1 1] - Predict = [[0.16215503]] - Label = [0]

## L2E5

weightsLayer1 = [[2.44721954 1.46061549]

[2.46002032 1.46289343]]

biasesLayer1 = [[-0.83922639 -2.00290008]]

weightsLayer2 = [[ 2.51271617]

[-2.49784305]]

biasLayer2 = [[-0.68695816]]

Input = [0 0] - Predict = [[-0.00067158]] - Label = [0]

Input = [0 1] - Predict = [[0.99356634]] - Label = [1]

Input = [1 0] - Predict = [[0.99356104]] - Label = [1]

Input = [1 1] - Predict = [[0.01038969]] - Label = [0]

## L2E6

weightsLayer1 = [[ 7.34644119 7.21733565]

[ 4.14064702 -3.13760731]]

biasesLayer1 = [[-0.8119718 2.28981795]]

weightsLayer2 = [[ 5.5062057 ]

[-4.96236505]]

biasLayer2 = [[-0.54302076]]

Input = [0 0] - Predict = [[0.03369915]] - Label = [0]

Input = [0 1] - Predict = [[0.96389219]] - Label = [1]

Input = [1 0] - Predict = [[0.49830063]] - Label = [1]

Input = [1 1] - Predict = [[0.50229456]] - Label = [0]