Rezolvarea problemelor cu ajutorul metodelor de învățare

Obiective

Dezvoltarea sistemelor care învaţă singure. Algoritmi de învăţare. Specificarea, proiectarea şi implementarea sistemelor care învaţă singure cum să rezolve probleme de clasificare.

Aspecte teoretice

Proiectarea şi dezvoltarea sistemelor care învaţă singure.
Algoritmi de învăţare de tipul:

- *stocastic gradient descent si retele neuronale*

Probleme abordate

1. *Remember* problema de regresie
    a. ce se da (input X, output Y, un input xnou), ce se cere (functia care transforma X in Y: f(X) = Y, astfel incat sa poata fi calculat ynou=f(xnou))
    b. ce poate fi X ? -->
        i. o lista de valori numerice (regresie simpla) X = (x1), x1 = x11, x21, ..., xn1), unde n e nr de exemple de antrenare),
        ii. vector cu mai multe dimensiuni de valori numerice (regresie multipla): daca avem 2 dimensiuni: X = (x1, x2), x1 = (x11, x21, ..., xn1), x2=(x12, x22, x32, ..., xn2), unde n e nr de exemple de antrenare
    c. ce poate fi Y? -->
        i. o lista de valori (pt un exemplu, trebuie prezis un singur output), Y = (y1), y1 = y11, y21, ..., yn1), unde n e nr de exemple de antrenare),
        ii. vector cu mai multe dimensiuni de valori: daca avem 3 dimensiuni: Y = (y1, y2, y3), y1 = (y11, y21, ..., yn1), y2=(y12, y22, y32, ..., yn2), y3 = (y13, y23, ..., yn3), unde n e nr de exemple de antrenare (pt un exemplu, trebuie prezise mai multe (3) output-uri)
2. Metode de identificare a functiei f - Rețele neuronale artificiale
3. Problemă
   Se cunosc următoarele informaţii pentru o perioadă de timp trecută: nivelul umidităţii - U, nivelul radiaţiilor solare - RS, intensitatea vântului – V – şi consumul orar de energie electrică – EE (datele normalizate aferente unui set de 10 înregistrări se găsesc în Tabel 1). Să se estimeze consumul orar de energie electrică pentru un tuplu de informaţii (umiditate=0.31, radiaţii solare = 0.55, intensitate vânt=0.82).

U	RS	V	EE
0.74	0.42	0.97	-0.33911
0.04	0.76	0.79	-0.73327
0.72	0.89	0.13	1.1539
0.13	0.26	0.14	-0.07017
0.65	0.49	0.79	-0.14347
0.43	0.44	0.70	-0.31482
0.86	0.68	0.99	0.17052

0.73	0.39	0.29	0.27971
0.08	0.96	0.56	-0.41447
0.47	0.12	0.72	-0.60652

Tabel 1 Date normalizate privind nivelul umidităţii, nivelul radiaţiilor solare şi intensitatea vântului

Încercaţi să rezolvaţi problema folosind o RNA cu următoarea structură:

- cu un strat de intrare cu 3 neuroni
- cu un singur strat ascuns care conţine 2 noduri, cu funcţia de activare liniara/sigmoid
- cu un singur neuron pe stratul de ieşire

Antrenarea reţelei

- rata de învăţare η=0.001
- limite pentru ponderile initiale: [0,1]
- nr epoci = 10
- neuroni cu activare liniara/sigmoid

```
class Neuron:
    def __init__(self, w = [], out = None, delta = 0.0):
        self.weights = w
        self.output = out
        self.delta = delta
    def __str__(self):
        return "weights: " + str(self.weights) + ", output: " + str(self.output) + ", delta: " + str(self.delta)
    def __repr__(self):
        return "weights: " + str(self.weights) + ", output: " + str(self.output) + ", delta: " + str(self.delta)

#initialisation of the weights for each neuron of all the layers (input layer & hidden layers)
def netInitialisation(noInputs, noOutputs, noHiddenNeurons):
    net = []
    '''hiddenLayer = []
    for h in range(noHiddenNeurons): #create hidden layers
        weights = [ random() for i in range(noInputs + 1)]    #noInputs and the bias
        neuron = Neuron(weights)
        hiddenLayer.append(neuron)'''
    hiddenLayer = [Neuron([ random() for i in range(noInputs + 1)]) for h in range(noHiddenNeurons)]
    net.append(hiddenLayer)
    '''outputLayer = []
    for o in range(noOutputs):
        weights = [ random() for i in range(noHiddenNeurons + 1)]
        neuron = Neuron(weights)
        outputLayer.append(neuron) '''
    outputLayer = [Neuron([ random() for i in range(noHiddenNeurons + 1)]) for o in range(noOutputs)]
    net.append(outputLayer)
    return net

def activate(input, weights):
    result = 0.0
    for i in range(0, len(input)):
        result += input[i] * weights[i]
    result += weights[len(input)]
    return result

#neuron transfer
def transfer(value):
    if (ACTIVATION == "Linear"):
        return value
    elif (ACTIVATION == "Sigmoid"):
        return 1.0 / (1.0 + exp(-value))

#neuron computation/activation
def forwardPropagation(net, inputs):
    for layer in net:
        newInputs = []
        for neuron in layer:
            activation = activate(inputs, neuron.weights)
```

```
            neuron.output = transfer(activation)
            newInputs.append(neuron.output)
        inputs = newInputs
    return inputs

#inverse transfer of a neuron
def transferInverse(val):
    if (ACTIVATION == "Linear"):
        return val
    elif (ACTIVATION == "Sigmoid"):
        return val * ( 1 - val)

#error propagation
def backwardPropagation(net, expected):
    for i in range(len(net) - 1, 0, -1):
        crtLayer = net[i]
        errors = []
        if (i == len(net) - 1): #last layer
            for j in range(0, len(crtLayer)):
                crtNeuron = crtLayer[j]
                errors.append(expected[j] - crtNeuron.output)
        else:   #hidden layers
            for j in range(0, len(crtLayer)):
                crtError = 0.0
                nextLayer = net[i + 1]
                for neuron in nextLayer:
                    crtError += neuron.weights[j] * neuron.delta
                errors.append(crtError)
        for j in range(0, len(crtLayer)):
            crtLayer[j].delta = errors[j] * transferInverse(crtLayer[j].output)

#change the weights
def updateWeights(net, example, learningRate):
    for i in range(0, len(net)):    #for each layer
        inputs = example[:-1]
        if (i > 0): #hidden layers or output layer
            inputs = [neuron.output for neuron in net[i - 1]]   #computed values of precedent layer
        for neuron in net[i]:   #update weight of all neurons of the current layer
            for j in range(len(inputs)):
                neuron.weights[j] += learningRate * neuron.delta * inputs[j]
            neuron.weights[-1] += learningRate * neuron.delta


def trainingMLP(net, data, noOutputTypes, learningRate, noEpochs):
    global PROBLEMTYPE
    for epoch in range(0, noEpochs):
        sumError = 0.0
        for example in data:
            inputs = example[:- 1]
            computedOutputs = forwardPropagation(net, inputs)
            if (PROBLEMTYPE == "classification"):
                expected = [0 for i in range(noOutputTypes)]
                expected[example[-1]] = 1
                computedLabels = [0 for i in range(noOutputTypes)]
                computedLabels[computedOutputs.index(max(computedOutputs))] = 1
                computedOutputs = computedLabels
            elif (PROBLEMTYPE == "regression"):
                expected = [example[-1]]
            crtErr = sum([(expected[i] - computedOutputs[i]) ** 2 for i in range(0, len(expected))])
            #print("Epoch: ", epoch, " example: ", example, " expected: ", expected, " computed: ", 
computedOutputs, " crtErr: ", crtErr)
            sumError += crtErr
            backwardPropagation(net, expected)
            updateWeights(net, example, learningRate)


def evaluatingMLP(net, data, noOutputTypes):
    computedOutputs = []
    for inputs in data:
        computedOutput = forwardPropagation(net, inputs[:-1])

        if (PROBLEMTYPE == "classification"):
            computedLabels = [0 for i in range(noOutputTypes)]
            computedLabels[computedOutput.index(max(computedOutput))] = 1
            computedOutput = computedLabels
        elif (PROBLEMTYPE == "regression"):
            pass
        computedOutputs.append(computedOutput[0])
```

```
    return computedOutputs


def computePerformanceRegression(computedOutputs, realOutputs):
    error = sum([(computedOutputs[i] - realOutputs[i]) ** 2 for i in range(len(computedOutputs))])
    return error


def computePerformanceClassification(computedOutputs, realOutputs):
    noOfMatches = sum([computedOutputs[i] == realOutputs[i] for i in range(0, len(computedOutputs))])
    return noOfMatches / len(computedOutputs)


def runMLP(trainData, testData, learningRate, noEpochs):
    global PROBLEMTYPE
    noInputs = len(trainData[0]) - 1
    if (PROBLEMTYPE == "classification"):
        noOutputs = len(set([example[-1] for example in trainData])) #for classification noOutputs = # of classes
        net = netInitialisation(noInputs, noOutputs, 2)

        trainingMLP(net, trainData, noOutputs, learningRate, noEpochs)


        realOutputs = [trainData[i][j] for j in range(len(trainData[0]) - 1, len(trainData[0]) )  for i in range(
0, len(testData) )]
        computedOutputs = evaluatingMLP(net, trainData[:-1], noOutputs)[0]
        print("train Acc: ", computePerformanceClassification(computedOutputs, realOutputs))

        realOutputs = [testData[i][j] for j in range(len(testData[0]) - 1, len(testData[0]) )  for i in range( 0,
len(testData) )]
        computedOutputs = evaluatingMLP(net, testData[:-1], noOutputs)[0]
        print("test Acc: ", computePerformanceClassification(computedOutputs, realOutputs))

    elif (PROBLEMTYPE == "regression"):
        noOutputs = 1
        net = netInitialisation(noInputs, noOutputs, 2)

        trainingMLP(net, trainData, noOutputs, learningRate, noEpochs)

        realOutputs = [trainData[i][j] for j in range(len(trainData[0]) - 1, len(trainData[0]) )  for i in range(
0, len(trainData) )]
        computedOutputs = evaluatingMLP(net, trainData, noOutputs)
        print("train SRE: ", computePerformanceRegression(computedOutputs, realOutputs))

        realOutputs = [testData[i][j] for j in range(len(testData[0]) - 1, len(testData[0]) )  for i in range( 0,
len(testData) )]
        computedOutputs = evaluatingMLP(net, testData, noOutputs)
        print("test SRE: ", computePerformanceRegression(computedOutputs, realOutputs))




def runMLP_tool(regressionDataTrain, regressionDataTest, learningRate, noEpochs):
    activationType = ""
    if ACTIVATION == "Linear":
        activationType = "identity"
    elif ACTIVATION == "Sigmoid":
        activationType = "logistic"

    ann = []
    if (PROBLEMTYPE == "regression"):
        ann = neural_network.MLPRegressor((2,), activationType, "sgd")
    elif (PROBLEMTYPE == "classification"):
        ann = neural_network.MLPClassifier((2,), activationType, "sgd")

    inputTrain = np.array([line[:-1] for line in regressionDataTrain])
    outputTrain = np.array([line[-1] for line in regressionDataTrain])
    ann.fit(inputTrain, outputTrain)
    computedOutputs = ann.predict(inputTrain)
    print("TOOL, train SRE: ", computePerformanceRegression(computedOutputs, outputTrain))

    inputTest = np.array([line[:-1] for line in regressionDataTest])
    outputTest = np.array([line[-1] for line in regressionDataTest])
    computedOutputs = ann.predict(inputTest)
    print("TOOL, test SRE: ", computePerformanceRegression(computedOutputs, outputTest))
```

```
def regression():
    #y = 2*x1 + x2 - x3
    regressionDataTrain = [[0.5, 0.05, 0.1, 0.95], [0.5, 0.1, 0.5, 0.6], [0.1, 0.2, 0.2, 0.2], [0.2, 0.3, 0.1, 
0.6], [0.3, 0.3, 0.1, 0.8]]
    regressionDataTest = [[0.2, 0.4, 0.1, 0.7], [0.2, 0.3, 0.5, 0.2], [0.3, 0.3, 0.7, 0.2]]

    regressionDataTrain2 = [[0.74, 0.42, 0.97, -0.33911],
                            [0.04, 0.76, 0.79, -0.73327],
                            [0.72, 0.89, 0.13, 1.1539],
                            [0.13, 0.26, 0.14, -0.07017],
                            [0.65, 0.49, 0.79, -0.14347],
                            [0.43, 0.44, 0.70, -0.31482],
                            [0.86, 0.68, 0.99, 0.17052],
                            [0.73, 0.39, 0.29, 0.27971],
                            [0.08, 0.96, 0.56, -0.41447],
                            [0.47, 0.12, 0.72, -0.60652]]
    regressionDataTest2 = [[0.31, 0.55, 0.82, 0.80]]

    PROBLEMTYPE = "regression"
    #ACTIVATION = "Linear"
    ACTIVATION = "Sigmoid"
    learningRate = 0.001
    noEpochs = 200

    runMLP(regressionDataTrain, regressionDataTest2,learningRate, noEpochs)
    runMLP_tool(regressionDataTrain, regressionDataTest, learningRate, noEpochs)


def classification():
    classificationDataTrain = [ [2.7810836,2.550537003,0],
                                [1.465489372,2.362125076,0],
                                [6.922596716,1.77106367,1],
                                [1.38807019,1.850220317,0],
                                [3.06407232,3.005305973,0],
                                [8.675418651,-0.242068655,1],
                                [7.627531214,2.759262235,1]]
    classificationDataTest =  [ [5.332441248,2.088626775,1],
                                [3.396561688,4.400293529,0],
                                [7.673756466,3.508563011,1]]

    PROBLEMTYPE = "classification"
    ACTIVATION = "Sigmoid"
    learningRate = 0.001
    noEpochs = 200

    runMLP(classificationDataTrain, classificationDataTest,learningRate, noEpochs)
    runMLP_tool(classificationDataTrain, classificationDataTest, learningRate, noEpochs)



regression()
#classification()
```