School of Data and Computer Science, Sun Yat-sen University

E14 BP Algorithm (C++/Python)

17341111 Xuehai Liu

2019 年 12 月 15 日

目录

1 Horse Colic Data Set

The description of the horse colic data set (`http://archive.ics.uci.edu/ml/datasets/Horse+Colic`) is as follows:

We aim at trying to predict if a horse with colic will live or die.

Note that we should deal with missing values in the data! Here are some options:

- Use the feature’ s mean value from all the available data.
- Fill in the unknown with a special value like -1.
- Ignore the instance.
- Use a mean value from similar items.
- Use another machine learning algorithm to predict the value.

2 Reference Materials

1. Stanford: **CS231n: Convolutional Neural Networks for Visual Recognition** by Fei-Fei Li,etc.
   - Course website: `http://cs231n.stanford.edu/2017/syllabus.html`
   - Video website: `https://www.bilibili.com/video/av17204303/?p=9&tdsourcetag=s_pctim_aiomsg`
2. **Machine Learning** by Hung-yi Lee
   - Course website: `http://speech.ee.ntu.edu.tw/~tlkagk/index.html`
   - Video website: `https://www.bilibili.com/video/av9770302/from=search`
3. A Simple neural network code template

```python
# -*- coding: utf-8 -*
import random
import math

# Shorthand:
# "pd_" as a variable prefix means "partial derivative"
# "d_" as a variable prefix means "derivative"
# "_wrt_" is shorthand for "with respect to"
# "w_ho" and "w_ih" are the index of weights from hidden to output layer neurons and input to hidden layer neurons respectively

class NeuralNetwork:
    LEARNING_RATE = 0.5
    def __init__(self, num_inputs, num_hidden, num_outputs, hidden_layer_weights = None, hidden_layer_bias = None, output_layer_weights = None, output_layer_bias = None):
    #Your Code Here

```

```python
    def init_weights_from_inputs_to_hidden_layer_neurons(self, hidden_layer_weights):
    #Your Code Here

    def init_weights_from_hidden_layer_neurons_to_output_layer_neurons(self, output_layer_weights):
    #Your Code Here

    def inspect(self):
        print('------')
        print('* Inputs: {}'.format(self.num_inputs))
        print('------')
        print('Hidden Layer')
        self.hidden_layer.inspect()
        print('------')
        print('* Output Layer')
        self.output_layer.inspect()
        print('------')

    def feed_forward(self, inputs):
        #Your Code Here

    # Uses online learning, ie updating the weights after each training case
    def train(self, training_inputs, training_outputs):
        self.feed_forward(training_inputs)

        # 1. Output neuron deltas
        #Your Code Here
        #  E/ z

        # 2. Hidden neuron deltas
        # We need to calculate the derivative of the error with respect to the output of each hidden layer neuron
        # dE/dy = Σ  E/ z  *  z/ y = Σ  E/ z  * w
        #  E/ z = dE/dy  *  z /
        #Your Code Here

        # 3. Update output neuron weights
        #  E / w  =  E/ z  *  z / w
        # Δw =   *  E / w
        #Your Code Here

        # 4. Update hidden neuron weights
        #  E / w =  E/ z  *  z / w
        # Δw =   *  E / w
        #Your Code Here

```

```python
    def calculate_total_error(self, training_sets):
        #Your Code Here
        return total_error

class NeuronLayer:
    def __init__(self, num_neurons, bias):

        # Every neuron in a layer shares the same bias
        self.bias = bias if bias else random.random()

        self.neurons = []
        for i in range(num_neurons):
            self.neurons.append(Neuron(self.bias))

    def inspect(self):
        print('Neurons:', len(self.neurons))
        for n in range(len(self.neurons)):
            print(' Neuron', n)
            for w in range(len(self.neurons[n].weights)):
                print('  Weight:', self.neurons[n].weights[w])
            print('  Bias:', self.bias)

    def feed_forward(self, inputs):
        outputs = []
        for neuron in self.neurons:
            outputs.append(neuron.calculate_output(inputs))
        return outputs

    def get_outputs(self):
        outputs = []
        for neuron in self.neurons:
            outputs.append(neuron.output)
        return outputs

class Neuron:
    def __init__(self, bias):
        self.bias = bias
        self.weights = []

    def calculate_output(self, inputs):
    #Your Code Here

    def calculate_total_net_input(self):
    #Your Code Here

    # Apply the logistic function to squash the output of the neuron
    # The result is sometimes referred to as 'net' [2] or 'net' [1]
```

```
    def squash(self, total_net_input):
    #Your Code Here

    # Determine how much the neuron's total input has to change to move closer to the expected output
    #
    # Now that we have the partial derivative of the error with respect to the output ( E/ y ) and
    # the derivative of the output with respect to the total net input (dy/dz ) we can calculate
    # the partial derivative of the error with respect to the total net input.
    # This value is also known as the delta ( ) [1]
    #   = E/ z = E/ y * dy/dz
    #
    def calculate_pd_error_wrt_total_net_input(self, target_output):
    #Your Code Here

    # The error for each neuron is calculated by the Mean Square Error method:
    def calculate_error(self, target_output):
    #Your Code Here

    # The partial derivate of the error with respect to actual output then is calculated by:
    # = 2 * 0.5 * (target output - actual output) ^ (2 - 1) * -1
    # = -(target output - actual output)
    #
    # The Wikipedia article on backpropagation [1] simplifies to the following, but most other learning material does not [2]
    # = actual output - target output
    #
    # Alternative, you can use (target - output), but then need to add it during backpropagation [3]
    #
    # Note that the actual output of the output neuron is often written as y  and target output as t  so:
    # =  E/ y  = -(t  - y )
    def calculate_pd_error_wrt_output(self, target_output):
    #Your Code Here

    # The total net input into the neuron is squashed using logistic function to calculate the neuron's output:
    # y  =   = 1 / (1 + e^(-z ))
    # Note that where   represents the output of the neurons in whatever layer we're looking at and   represents the layer below it
    #
    # The derivative (not partial derivative since there is only one variable) of the output then is:
```

```
    # dy/dz = y * (1 - y )
    def calculate_pd_total_net_input_wrt_input(self):
    #Your Code Here

    # The total net input is the weighted sum of all the inputs to the neuron and their respective weights:
    # = z = net = x w + x w ...
    #
    # The partial derivative of the total net input with respective to a given weight (with everything else held constant) then is:
    # = z / w = some constant + 1 * x w ^(1-0) + some constant ... = x
    def calculate_pd_total_net_input_wrt_weight(self, index):
    #Your Code Here

# An example:

nn = NeuralNetwork(2, 2, 2, hidden_layer_weights=[0.15, 0.2, 0.25, 0.3], hidden_layer_bias=0.35, output_layer_weights=[0.4, 0.45, 0.5, 0.55], output_layer_bias=0.6)
for i in range(10000):
    nn.train([0.05, 0.1], [0.01, 0.99])
    print(i, round(nn.calculate_total_error([[[0.05, 0.1], [0.01, 0.99]]]), 9))
```

3 Tasks

- Given the training set `horse-colic.data` and the testing set `horse-colic.test`, implement the BP algorithm and establish a neural network to predict if horses with colic will live or die. In addition, you should calculate the accuracy rate.
- Please submit a file named `E14_YourNumber.pdf` and send it to `ai_201901@foxmail.com`

4 Codes and Results

Firstly, there goes the code of a layer. I simply define a layer including the function of calculating derivatives, output and input of a layer, and also a activative function sigmoid.

```
import math
import numpy as np

class Layer:
    def __init__(self, input=[]):
        self.input = input
        self.output = []
        self.output = self.calculate_output(input)
        self.input_deltas = []  # 指每个神经元输入值的误差

    def calculate_output(self, input):
        self.output.clear()
        self.output.append(1)
        for neuron in input:
            self.output.append(self.squash(neuron))
        return self.output

    def calculate_input_delta(self, weight, post_delta):
        weight_matrix = np.array(weight)
        post_delta_matrix = np.array(post_delta)
        output_matrix = np.array(self.output)
        delta_matrix = weight_matrix.dot(post_delta_matrix) * output_matrix * (1 -
                                                            output_matrix)
        self.input_deltas = delta_matrix.tolist()
        return self.input_deltas

    def set_output(self, output):
        self.output = output

    def set_input(self, input):
        self.input = input

    def set_input_deltas(self, delta):
        self.input_deltas = delta

    def get_output(self):
        return self.output

    def get_input(self):
        return self.input

    def get_input_deltas(self):
        return self.input_deltas

```

```python
    # 激活函数sigmod
    def squash(self, input):
        return 1 / (1 + math.exp(-input))
```

Secondly, using the definition above, construct the neural network:

Please note that I introduce a parameter 'lambda' as a weight attenuation parameter, to avoid over fitting.

```python
class NeuralNetwork:
    def __init__(self):
        self.weight = []
        self.layers = []
        self.weight_deltas = []

    # 初始化权重矩阵
    def init_weight(self, input_dim, hidden_dim, hidden_num):
        # input_dim + 1是因为输出层有一个偏置单元，下面同理
        self.weight.append((2 * INIT_EPSILON) * np.random.rand(input_dim + 1, hidden_dim) - INIT_EPSILON)
        for i in range(hidden_num - 1):
            self.weight.append((2 * INIT_EPSILON) * np.random.rand(hidden_dim + 1, hidden_dim) - INIT_EPSILON)
        self.weight.append((2 * INIT_EPSILON) * np.random.rand(hidden_dim + 1, OUTPUT_DIM) - INIT_EPSILON)

    # 基于反向传播算法的神经网络学习
    def back_propagation(self, training_data, hidden_dim, hidden_num):
        input_dim = training_data[0][1].__len__()
        training_num = training_data.__len__()
        self.init_weight(input_dim, hidden_dim, hidden_num)

        #一共有hidden_num+2层，一层输入一层输出。
        for i in range(hidden_num + 2):
            self.layers.append([])
        for times in range(ITERATION):
            if(times % 100 == 0):
                print('第', times + 1, '次迭代')

            #初始化误差
            self.init_deltas(input_dim, hidden_dim, 1, hidden_num)
            cost = 0
            sum = 0

            for y, x in training_data:
                # 正向传播，先构造输入层
                input_layer = layer.Layer(x)  # 将训练样本值放入输入层
                input_layer.set_output([1] + x)  # 输入层的输出值等于输入值，不必计算激活函数值，但需要增加一个偏差单元
                self.layers[0] = input_layer

```

```
                # 构造隐藏层和输出层
                for i in range(1, hidden_num + 2):
                    hidden_input = np.dot(self.layers[i - 1].get_output(), self.weight[i - 1])  # 根据上一层的输出计算本层输入值
                    hidden_layer = layer.Layer(hidden_input)  # 根据输入构造一个新的隐藏层
                    self.layers[i] = hidden_layer

                # 获得输出神经元的值
                output = self.layers[hidden_num + 1].get_output()  # 因为输出层里面包括了偏差单元，所以output[1]才是输出神经元
                cost += (y * math.log(output[1]) + (1 - y) * math.log(1 - output[1])) # 计算输出与实际的损失
                #print('predict:', output[1], 'actual:', y)

                sum += ( abs(output[1]-y) <0.25)    #判决函数

                # 反向传播
                self.layers[hidden_num + 1].set_input_deltas([(output[1] - y) * output[1] * (1 - output[1])])  # 先算出输出层输入值的误差

                # 计算隐藏层和输入层的神经元输入值误差
                for i in range(hidden_num, -1, -1):
                    post_delta = self.layers[i + 1].get_input_deltas()

                    #去除偏置
                    if i != hidden_num:
                        del post_delta[0]
                    input_deltas = self.layers[i].calculate_input_delta(self.weight[i], post_delta)     #计算偏导数
                    self.layers[i].set_input_deltas(input_deltas)

                # 计算隐藏层和输出层的权重误差，并累加到weight_deltas上
                for i in range(hidden_num + 1):
                    self.weight_deltas[i] = NeuralNetwork.calculate_weight_delta(self.weight_deltas[i],
                                                                                 self.
```

```
                                                                                                                                  self.

            if(times % 100)== 0:
                acc = sum/len(training_data)
                print("train acc:",acc)

            # 以所有样本权重误差累计值平均值作为偏导值，调整权重
            for l in range(hidden_num + 1):
                for i in range(self.weight[l].__len__()):
                    for j in range(self.weight[l][i].__len__()):
                        if i == 0:
                            self.weight_deltas[l][i][j] = self.weight_deltas[l][i][j] / training_num
                        else:
                            self.weight_deltas[l][i][j] = (self.weight_deltas[l][i][j] + LAMBDA * self.weight[l][i][
                                j]) / training_num
                            #lambda是权重衰减参数，防止过拟合。

                        #梯度下降
                        self.weight[l][i][j] = self.weight[l][i][j] - ALPHA * self.weight_deltas[l][i][j]



    # 初始化偏导矩阵为零矩阵
    def init_deltas(self, input_dim, hidden_dim, output_dim, hidden_num):
        # 初始化偏导矩阵
        self.weight_deltas = []
```

```python
        self.weight_deltas.append(np.zeros([input_dim + 1, hidden_dim]))
        for i in range(hidden_num - 1):
            self.weight_deltas.append(np.zeros([hidden_dim + 1, hidden_dim]))
        self.weight_deltas.append(np.zeros([hidden_dim + 1, output_dim]))

    # 计算对权重的偏导数

    #self.weight_deltas[i], self.layers[i].get_output(), self.layers[i + 1].get_input_deltas()

    #根据第i层的权重残差和第i层的输出与其上一层的输入残差计算第i层的权重偏导数
    @staticmethod
    def calculate_weight_delta(weight_delta, output, post_deltas):
        weight_delta_matrix = np.array(weight_delta)
        output_matrix = np.array(output).reshape((1, output.__len__()))
        post_deltas_matrix = np.array(post_deltas).reshape((1, post_deltas.__len__()))
        res = weight_delta_matrix + output_matrix.T.dot(post_deltas_matrix)
        return res

    def predict(self, x):
        input_layer = layer.Layer(x)  # 将训练样本值放入输入层
        input_layer.set_output([1] + x)  # 输入层的输出值等于输入值，不必计算激活函数值，但需要增加一个偏差单元
        self.layers[0] = input_layer

        # 构造隐藏层和输出层
        for i in range(1, hidden_num + 2):
            hidden_input = np.dot(self.layers[i - 1].get_output(), self.weight[i - 1])  # 根据上一层的输出计算本层输入值
            hidden_layer = layer.Layer(hidden_input)  # 根据输入构造一个新的隐藏层
            self.layers[i] = hidden_layer

        # 获得输出神经元的值
        output = self.layers[hidden_num + 1].get_output()  # 输出层的第一个单元是偏差单元，output[1]才是输出神经元
        return output[1]
```

Thirdly, preprocess the data. In the file predeal.py, I define several functions to preprocess the data. I used one-hot encoding to deal with the discrete attributes, and apply normalization to the continious data. Please note that the attribute from colume 25 - 27 is not correct in the data, and therefore I did not use them. The final input vector is flatted and the length is 67.

```
import pandas as pd
import numpy as np

def createDataSet(path):
    dataset = []
    with open(path,encoding = 'utf-8') as datafile:
        for line in datafile.readlines():
            list = line.split()
            for i,str in enumerate(list):
                if(str != '?'):
                    list[i] = float(str)
            dataset.append(list)
    return dataset

#第一步构建初始数据集
train_data = createDataSet("horse-colic.data")
test_data = createDataSet("horse-colic.test")


#此函数将数据集中的？用该列的平均值替代。
def deallabel(dataset,label):
    list = [dataset[i][label] for i in range(len(dataset))]
    avg = 0
    sum = 0
    num = 0
    for i in list:
        if(i != '?'):
            sum += i
            num += 1
    avg = sum/num
    for i,str in enumerate(list):
        if(str == '?'):
            list[i] = avg
    for i in range(len(dataset)):
        dataset[i][label] = list[i]

def preDeal(dataset,filename):
    for i in range(len(dataset[0])):
        deallabel(dataset,i)
    import csv
    with open (filename,'w',newline= '')as f:
        writer = csv.writer(f)
```

```
        for line in dataset:
            writer.writerow(line)
        f.close()

preDeal(train_data,'horse_colic_deal.data')
preDeal(test_data,'horse_colic_deal.test')


print(train_data[:2])
print(test_data[:2])

def normalization(datingDatamat):
   max_arr = datingDatamat.max(axis=0)
   min_arr = datingDatamat.min(axis=0)
   ranges = max_arr - min_arr
   norDataSet = np.zeros(datingDatamat.shape)
   m = datingDatamat.shape[0]
   norDataSet = datingDatamat - np.tile(min_arr, (m, 1))
   norDataSet = norDataSet/np.tile(ranges,(m,1))
   return norDataSet

def createSubDataset(dataset, listlabels):
    newdataset = []
    for label in listlabels:
        # list = [dataset[i][label]for i in range(len(dataset))]
        list = [row[label] for row in dataset]
        # print(list)
        newdataset.append(list)
    return newdataset


def createFinalDataset(preDataset):
    normal = [3, 4, 5, 6, 16, 19, 20, 22]
    discrete = [1, 2, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 21, 23, 24, 28]
    output = 23
    normal = [i - 1 for i in normal]
    discrete = [i - 1 for i in discrete]

    discreteDataset = (np.mat(createSubDataset(preDataset, discrete))).T
    continiousDataset = (np.mat(createSubDataset(preDataset, normal))).T

    continiousDataset = normalization(continiousDataset)

    from sklearn import preprocessing
    enc = preprocessing.OneHotEncoder()
    enc.fit(discreteDataset)

```

```
    print(continiousDataset[1].tolist())
    e = enc.transform(discreteDataset[1]).toarray()
    print(e)

    newdataset = []
    for i in range(len(discreteDataset)):
        list1 = continiousDataset[i].tolist()[0]
        list2 = enc.transform(discreteDataset[i]).toarray().tolist()[0]
        newdataset.append(list1 + list2)

    labels = [(row[22] - 1) / 2 for row in preDataset]
    dataset = []

    for i in range(len(labels)):
        tuple = (labels[i], newdataset[i])
        dataset.append(tuple)

    return dataset
```

Finally, test the network on the test data. Main function does the job and the code is as follow. Please note that because I simply use one output node, so that the judge function is simple: answers of 1, 2, and 3 are normalized to 0, 0.5 and 1, we choose the closest one to the predict value to be the predicted answer.

By the way, the structure of network is 67 - 5 - 1 in this case.

```

if __name__ == '__main__':
    output_dim = 1
    hidden_dim = 5
    hidden_num = 1

    def createDataset(filename):
        pan = pd.read_csv(filename, header=None)
        matrix = pan.as_matrix().tolist()
        dataset = []
        for line in matrix:
            for i, num in enumerate(line):
                line[i] = round(num,2)
            dataset.append(line)
        return dataset

    training_data = createDataset("horse_colic_deal.data")

    testing_data = createDataset("horse_colic_deal.test")

    label_list = [testing_data[i][22] for i in range(len(testing_data))]

    combinedDataset = []
    for line in training_data:
        combinedDataset.append(line)
    for line in testing_data:
        combinedDataset.append(line)

    #此处将train和test数据集合并的目的是保证他们拥有相同的one-hot编码
    combinedDataset = createFinalDataset(combinedDataset)
    train_data = combinedDataset[:301]
    print(len(train_data[0][1]))

    test_data = combinedDataset[301:]

    #train_data = createFinalDataset(training_data)
    #test_data = createFinalDataset(testing_data)

    network = NeuralNetwork()

    network.back_propagation(train_data, hidden_dim, hidden_num)
```

```
    sum = 0
    for line in test_data:
        label = line[0]
        x = line[1]
        output = network.predict(x)
        print("output: {} , label: {}".format( output,label))
        sum += (abs(output- label) < 0.25)
    acc = sum / len(test_data)
    print("acc: {}".format(acc))
```

The final result seems to be very satisfying. After training 6000 iterations, the accuracy on test data has arrived at 0.94, and training 8000 iterations leads to a 1.00 accuracy on test data.

```
output: 0.477353775026128 , label: 0.5
output: 0.08769333524242562 , label: 0.0
output: 0.04690968496609765 , label: 0.0
output: 0.6779885922908904 , label: 1.0
output: 0.06963857019340153 , label: 0.0
output: 0.45514421017289763 , label: 0.5
acc: 0.9402985074626866
```

图 1: test result of 6000 training iterations

```
output: 0.5090170246581074 , label: 0.5
output: 0.07083245983639509 , label: 0.0
output: 0.04142735539902171 , label: 0.0
output: 0.8018693134727854 , label: 1.0
output: 0.05634973362181594 , label: 0.0
output: 0.4472685566480645 , label: 0.5
acc: 1.0

Process finished with exit code 0
```

图 2: test result of 8000 training iterations