SCHOOL OF DATA AND COMPUTER SCIENCE, SUN YAT-SEN UNIVERSITY

E14 BP Algorithm (C++/Python)

17341189 姚森舰

2019 年 12 月 13 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:

Data Set Characteristics:	Multivariate	**Number of Instances:**	368	**Area:**	Life
Attribute Characteristics:	Categorical, Integer, Real	**Number of Attributes:**	27	**Date Donated**	1989-08-06
Associated Tasks:	Classification	**Missing Values?**	Yes	**Number of Web Hits:**	108569

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

```
# -*- 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

    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

    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

4.1 Codes

```
# coding=utf-8
import numpy as np

```

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


# sigmoid的一阶导数
def sigmoid_prime(x):
    return sigmoid(x) * (1-sigmoid(x))


def onehot_encode(num, len):
    res = [0] * len
    res[num] = 1
    return res


class NeuralNetwork(object):
    def __init__(self, input_dim, hidden_node_num, output_dim):
        self.lr = 0.01                                          # 学习率
        self.input_dim = input_dim
        self.hidden_node_num = hidden_node_num
        self.output_dim = output_dim
        # 随机初始化
        self.w_ih = np.random.randn(input_dim, hidden_node_num) * 0.01
        self.b_ih = np.random.randn(hidden_node_num) * 0.01
        self.w_ho = np.random.randn(hidden_node_num, output_dim) * 0.01
        self.b_ho = np.random.randn(output_dim) * 0.01
        # 以下只是为了规范，在init()里面声明而已，初始值没有意义
        self.net_ih = np.zeros((hidden_node_num, 1))
        self.h_o = np.zeros((hidden_node_num, 1))
        self.net_ho = np.zeros((output_dim, 1))
        self.output = np.zeros((output_dim, 1))
        self.sensitivity_ho = np.zeros((output_dim, 1))
        self.x = np.zeros((input_dim, 1))

    # ===================================================== 关键代码部分
    # 前向传播
    def forward(self, x, y):
        self.x = x
        self.net_ih = np.dot(x, self.w_ih) + self.b_ih             # 隐藏层wx+b 一维向
            量相加都是对应元素相加
        self.h_o = sigmoid(self.net_ih)                            # 激活
        self.net_ho = np.dot(self.h_o, self.w_ho) + self.b_ho      # 输出层wx+b
        self.output = sigmoid(self.net_ho)                         # 激活
        loss = np.sum((self.output - y) * (self.output - y)) / 2
        self.sensitivity_ho = (self.output - y) * sigmoid_prime(self.output)     # 灵敏
            度，会用到所以保存
        return loss, self.output                                   # 方便预测
```

```
    # 反向传播
    def backward(self):
        # 10x3  =  10x1 dot 1x3      一维向量只会做内积，所以reshape为矩阵
        delta_w_ho = np.dot(self.h_o.reshape(self.hidden_node_num, 1),
                            self.sensitivity_ho.reshape(1, self.output_dim))
        delta_b_ho = self.sensitivity_ho                            # 偏置的输入为1
        sensitivity_ih = np.dot(self.sensitivity_ho, self.w_ho.T) * sigmoid_prime(self.
            net_ih)
        # 36x10 = 36x1 dot 1x10
        delta_w_ih = np.dot(self.x.reshape(self.input_dim, 1),
                            sensitivity_ih.reshape(1, self.hidden_node_num))
        delta_b_ih = sensitivity_ih                                 # 偏置的输入为1

        # 更新参数
        self.w_ho -= self.lr * delta_w_ho
        self.b_ho -= self.lr * delta_b_ho
        self.w_ih -= self.lr * delta_w_ih
        self.b_ih -= self.lr * delta_b_ih

        # p = np.random.random()
        # if p > 0.8:
        #     self.lr = self.lr * 0.9
    # ==================================================


def read_data(filename):
    """
    读取数据，并将分类标签放在最后一列
    :param filename:
    :return:
    """
    with open(filename, 'r') as f:
        data = f.readlines()
        dataset = []
        output_index = data[0].split(',').index('outcome')
        for row in data[1:]:
            row = row.split(',')
            row = list(map(float, row))
            outcome = int(row[output_index])
            row = row[:output_index] + row[output_index+1:]
            row.append(outcome)
            dataset.append(row)
        return dataset


def train(training_data, epoch, nn):
```

```
    for i in range(epoch):
        for data in training_data:
            x = np.array(data[:-1])
            y = onehot_encode(data[-1]-1, 3)
            y = np.array(y)
            loss, _ = nn.forward(x, y)
            nn.backward()
        print('Epoch {} Loss: {}'.format(i+1, loss))
    return nn


def test(testing_data, nn):
    """
    依次取数据喂入网络，得到输出结果与真正的标签对比
    :param testing_data:
    :param nn: 训练好的网络
    :return: 无
    """
    n = 0
    predictions = []
    for data in testing_data:
        x = np.array(data[:-1])
        y = [0, 0, 0]
        _, prediction = nn.forward(x, y)
        label = np.argmax(prediction, axis=0) + 1
        predictions.append(label)
        if label == data[-1]:
            n += 1
    print('Accuracy on testing dataset: {:.4}%'.format(100*n/len(testing_data)))
    print('Predictions:', predictions)


def feature_scaling(dataset):
    """
    对属性进行归一化处理
    :param dataset: 数据集
    :return: 归一化后的数据集
    """
    feature_num = len(dataset[0]) - 1
    maxs = [float('-inf')] * feature_num
    mins = [float('inf')] * feature_num
    res = []
    for data in dataset:
        for i in range(feature_num):
            if data[i] > maxs[i]:
                maxs[i] = data[i]
            if data[i] < mins[i]:
```

```
                mins[i] = data[i]
    # 归一化 使属性落到 [0,1]
    for data in dataset:
        for i in range(feature_num):
            if (maxs[i] - mins[i]) != 0:
                data[i] = (data[i] - mins[i]) / (maxs[i] - mins[i])
        res.append(data)
    return res


if __name__ == '__main__':
    training_data = read_data('horse-colic-data.csv')
    testing_data = read_data('horse-colic-test.csv')
    training_data = feature_scaling(training_data)
    testing_data = feature_scaling(testing_data)
    nn = NeuralNetwork(35, 10, 3)
    Epoch = 90
    nn = train(training_data, Epoch, nn)
    test(testing_data, nn)

    # for Epoch in range(50, 2000, 5):
    #     print('Epoch = {}'.format(Epoch))
    #     nn = NeuralNetwork(35, 10, 3)
    #     nn = train(training_data, Epoch, nn)
    #     test(testing_data, nn)
```

4.2 Results

运行结果如下:

```
Epoch 71 Loss: 0.6488004955851933
Epoch 72 Loss: 0.6508474674172009
Epoch 73 Loss: 0.6529190935811984
Epoch 74 Loss: 0.6550144297073128
Epoch 75 Loss: 0.6571324808494506
Epoch 76 Loss: 0.6592722046978403
Epoch 77 Loss: 0.6614325149180612
Epoch 78 Loss: 0.6636122845930651
Epoch 79 Loss: 0.6658103497477633
Epoch 80 Loss: 0.6680255129384165
Epoch 81 Loss: 0.6702565468912403
Epoch 82 Loss: 0.6725021981763113
Epoch 83 Loss: 0.6747611909040203
Epoch 84 Loss: 0.6770322304320285
Epoch 85 Loss: 0.6793140070709867
Epoch 86 Loss: 0.6816051997772608
Epoch 87 Loss: 0.6839044798206729
Epoch 88 Loss: 0.6862105144148678
Epoch 89 Loss: 0.688521970297488
Epoch 90 Loss: 0.6868375172469206
Accuracy on testing dataset: 76.47%
Predictions: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1]
```

一般准确率在 75% 左右, 较好的会有 78%, 实际上差距只有一俩个。在训练时，forword() 向前传播会修改网络的一些属性，而这些属性在反向传播过程中在训练时，forword() 向前传播会修改网络的一些属性，而这些属性在反向传播过程中会用到，所以我的实现不能边训练边测试，或者是做倒是可以这样做，但是不科学，因为这样 backward() 用到的数据不是训练的数据。主要是一开始没有考虑到这个地方。但是因为运行速度比较快，所以可以每次训练不同的 epoch，重新测试，可以看 main() 函数中注释掉的那几行代码，运行效果如下：

```
Epoch = 80
Accuracy on testing dataset: 70.59%
Epoch = 85
Accuracy on testing dataset: 72.06%
Epoch = 90
Accuracy on testing dataset: 76.47%
Epoch = 95
Accuracy on testing dataset: 75.0%
Epoch = 100
Accuracy on testing dataset: 75.0%
Epoch = 105
Accuracy on testing dataset: 75.0%
Epoch = 110
Accuracy on testing dataset: 73.53%
Epoch = 115
Accuracy on testing dataset: 73.53%
Epoch = 120
Accuracy on testing dataset: 72.06%
Epoch = 125
Accuracy on testing dataset: 70.59%
Epoch = 130
Accuracy on testing dataset: 72.06%
```

本次实验中，更新权值部分的矩阵运算比较复杂，容易出错，用笔算一下更容易理清思路。除此之外，实现的过程中主要遇到了两个问题，一是最开始数据没有归一化，导致预测结果全部为第 1

类。因为某些特征取值较大，如果不归一化的话，会在网络中占主导地位，相当于”存在偏见”，从而导致训练效果不好。另一个问题是，实现时犯了个小错误一开始读入数据后，想将 label 放到最后一列，也就是在第89行代码附近，拼接列表时应该是[output+1:]，写成了[output:]，导致标签也被当作了一个特征，最后出现了98%甚至100%的准确率，准确率高到让人不敢相信，好在与助教探讨后发现了问题。

因为助教建议用numpy实现，所以也就没有用给出的框架实现。