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

# E14 BP Algorithm (C++/Python)

20214966 Yangkai Lin 20214810 Suixin Ou

2020 年 12 月 11 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 = xw + xw ...
    #
    # 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 * xw^(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_2020@foxmail.com`
- Draw the training loss and accuracy curves
- (optional) You can try different structure of neural network and compare their accuracy and the time they cost.

## 4 Codes and Results

```
# -*- coding: utf-8 -*
import random
import math
import numpy as np
import matplotlib.pyplot as plt

EPOCH = 100
HIDDEN_NUM = 20
FUN = 0

def F(x):
    if FUN == 0:
        return sigmoid(x)
    elif FUN == 1:
```

```
            return relu(x)
        elif FUN == 2:
            return tanh(x)

    def d_F(x):
        if FUN == 0:
            return d_sigmoid(x)
        elif FUN == 1:
            return d_relu(x)
        elif FUN == 2:
            return d_tanh(x)

    def relu(x):
        return max(x, 0)

    def d_relu(x):
        return 1 if x > 0 else 0

    def sigmoid(x):
        return 1 / (1 + math.exp(-x))

    def d_sigmoid(x):
        return x * (1 - x)

    def tanh(x):
        return math.tanh(x)

    def d_tanh(x):
        return -x ** 2

    # 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

```

```python
class NeuralNetwork:
    LEARNING_RATE = 0.5

    # 这里默认一层隐藏层
    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
        self.num_inputs = num_inputs
        self.hidden_layer = NeuronLayer(num_hidden, hidden_layer_bias)
        self.output_layer = NeuronLayer(num_outputs, output_layer_bias)
        self.init_weights_from_inputs_to_hidden_layer_neurons(
            hidden_layer_weights)
        self.init_weights_from_hidden_layer_neurons_to_output_layer_neurons(
            output_layer_weights)

    def init_weights_from_inputs_to_hidden_layer_neurons(self,
        hidden_layer_weights):
        # Your Code Here
        if hidden_layer_weights is not None:
            for i in range(len(self.hidden_layer.neurons)):
                for j in range(self.num_inputs):
                    self.hidden_layer.neurons[i].weights.append(
                        hidden_layer_weights[i * self.num_inputs + j])
        else:
            for i in range(len(self.hidden_layer.neurons)):
                for j in range(self.num_inputs):
                    self.hidden_layer.neurons[i].weights.append(random.
                        random())
                    # self.hidden_layer.neurons[i].weights.append(.5)


    def init_weights_from_hidden_layer_neurons_to_output_layer_neurons(self,
         output_layer_weights):
        # Your Code Here
        if output_layer_weights is not None:
            for i in range(len(self.output_layer.neurons)):
```

```
                for j in range(len(self.hidden_layer.neurons)):
                    self.output_layer.neurons[i].weights.append(output_layer_weights[i * len(self.hidden_layer.neurons) + j])
        else:
            for i in range(len(self.output_layer.neurons)):
                for j in range(len(self.hidden_layer.neurons)):
                    self.output_layer.neurons[i].weights.append(random.random())
                    # self.output_layer.neurons[i].weights.append(.5)

    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
        return self.output_layer.feed_forward(self.hidden_layer.feed_forward(inputs))

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

        # 1. Output neuron deltas
        # E / z
        # Your Code Here
        output_delta = []
        for ind, i in enumerate(self.output_layer.neurons):
```

```
                output_delta.append(i.calculate_pd_error_wrt_total_net_input(training_outputs[ind]))

        # 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
        hidden_delta = []
        for ind1, i in enumerate(self.hidden_layer.neurons):
            sum = 0
            for ind2, j in enumerate(self.output_layer.neurons):
                sum += j.weights[ind1] * output_delta[ind2]
            hidden_delta.append(sum * d_F(self.hidden_layer.get_outputs()[ind1]))

        # 3. Update output neuron weights
        # E/w = E/z * z/w
        # Δw = * E/w
        # Your Code Here
        for i in range(len(self.output_layer.neurons)):
            for j in range(len(self.output_layer.neurons[i].weights)):
                self.output_layer.neurons[i].weights[j] += self.LEARNING_RATE * self.output_layer.neurons[i].calculate_pd_total_net_input_wrt_weight(i) * output_delta[i]

        # 4. Update hidden neuron weights
        # E/w = E/z * z/w
        # Δw = * E/w
        # Your Code Here
        for i in range(len(self.hidden_layer.neurons)):
            for j in range(len(self.hidden_layer.neurons[i].weights)):
                self.hidden_layer.neurons[i].weights[j] += self.LEARNING_RATE * self.hidden_layer.neurons[i].calculate_pd_total_net_input_wrt_weight(j) * hidden_delta
```

```
                        [i]
            # self.LEARNING_RATE *= 0.9999

    def calculate_total_error(self, training_sets):
        # Your Code Here
        total_error = 0
        for inputs, outputs in training_sets:
            self.feed_forward(inputs)
            for ind in range(len(outputs)):
                total_error += self.output_layer.neurons[ind].calculate_error(outputs[ind])
        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.bias = bias if bias else 0

        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.outputs)
        return outputs


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

    def calculate_output(self, inputs):
        # Your Code Here
        self.inputs = inputs
        self.outputs = self.squash(self.calculate_total_net_input())
        return self.outputs

    def calculate_total_net_input(self):
        # Your Code Here
        sum = 0
        for i in range(len(self.inputs)):
            sum += self.inputs[i] * self.weights[i]
        return sum + self.bias  # +b

    # 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
        return F(total_net_input)

    # 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
        return self.calculate_pd_error_wrt_output(target_output) * self.calculate_pd_total_net_input_wrt_input()

    # The error for each neuron is calculated by the Mean Square Error method:
    def calculate_error(self, target_output):
        # Your Code Here
        # 均方误差
        return 0.5 * (target_output - self.outputs) ** 2

    # 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
        # 均方误差
        return target_output - self.outputs

    # 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
        return d_F(self.outputs)

    # The total net input is the weighted sum of all the inputs to the neuron and their respective weights:
    # =  z = net = xw + xw ...
    #
    # 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 * xw^(1-0) + some constant ... = x
    def calculate_pd_total_net_input_wrt_weight(self, index):
        # Your Code Here
        return self.inputs[index]


def load(file_name):
    with open(file_name) as f:
        rst = []
        for i in f.readlines():
            curr = []
            for j in i[:-1].split('␣'):
                if j != '':
                    curr.append(j)
```

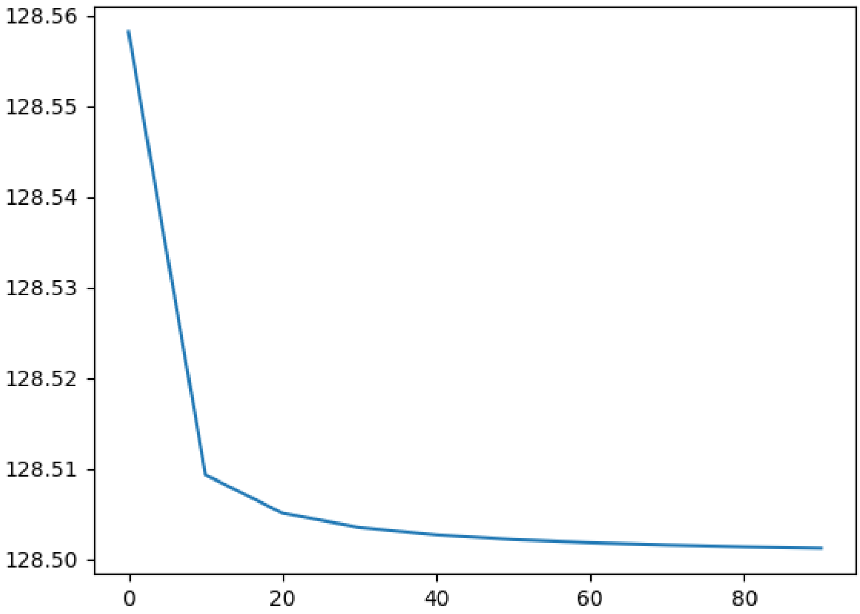
```
            rst.append(curr)
        return rst


def main():
    # 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(10):
        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))

    # load
    train = load("horse-colic.data")
    test = load("horse-colic.test")

    # wash
    # 改未知数据为 -1.0
    train_data = [i[:22] + i[23:-3] for i in train]
    train_label = [[float(i[22])] if i[22] != '?' else [-1.0] for i in train]
    test_data = [i[:22] + i[23:-3] for i in test]
    test_label = [[float(i[22])] if i[22] != '?' else [-1.0] for i in test]
    for i in range(len(train_data)):
        for j in range(len(train_data[i])):
            train_data[i][j] = float(train_data[i][j]) if train_data[i][j] != '?' else -1.0
    for i in range(len(test_data)):
        for j in range(len(test_data[i])):
                test_data[i][j] = float(test_data[i][j]) if test_data[i][j] != '?' else -1.0

    # 属性归一化
    for i in range(len(train_data[0])):
```

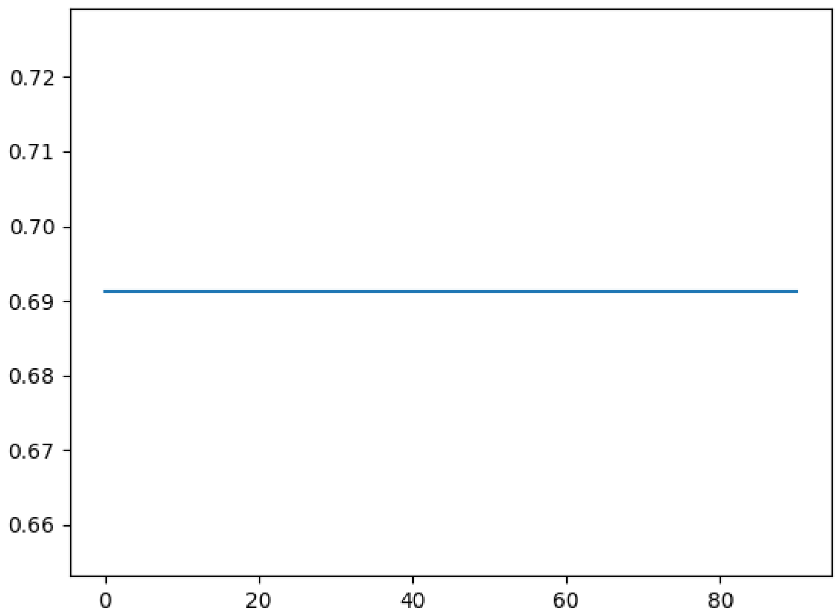
```
        mmax = max([k[i] for k in train_data])
        mmin = min([k[i] for k in train_data])
        max_min = mmax - mmin
        # j = max([math.log10(k[i]) if k[i]>0 else 0 for k in train_data])
        # mmean = np.mean([k[i] for k in train_data])
        # sstd = np.std([k[i] for k in train_data])
        for j in range(len(train_data)):
            # 最小最大规范化–
            # train_data[j][i] = (train_data[j][i] - mmin)/(max_min if max_min > 0 else (mmax if mmax else 1))
            train_data[j][i] = (train_data[j][i] - mmin)/max_min
            # 小数定标规范化
            # train_data[j][i] /= 10 ** j
            # 零均值规范化–
            # train_data[j][i] = (train_data[j][i] - mmean)/sstd

    for i in range(len(test_data[0])): #
        mmax = max([k[i] for k in test_data])
        mmin = min([k[i] for k in test_data])
        max_min = mmax - mmin
        # j = max([math.log10(k[i]) if k[i]>0 else 0 for k in test_data])
        # mmean = np.mean([k[i] for k in test_data])
        # sstd = np.std([k[i] for k in test_data])
        for j in range(len(test_data)):
            # 最小最大规范化–
            # test_data[j][i] = (test_data[j][i] - mmin)/(max_min if max_min > 0 else (mmax if mmax else 1))
            test_data[j][i] = (test_data[j][i] - mmin)/max_min
            # 小数定标规范化
            # test_data[j][i] /= 10 ** j
            # 零均值规范化–
            # test_data[j][i] = (test_data[j][i] - mmean) / sstd

    # # 标签归一化
    # mmax = max(train_label, key=lambda x: x[0])[0]
    # mmin = min(train_label, key=lambda x: x[0])[0]
    # max_min = mmax - mmin
```

```
    # train_label = list(map(lambda x: [(x[0] - mmin)/max_min], train_label)
        )
    #
    # mmax = max(test_label, key=lambda x: x[0])[0]
    # mmin = min(test_label, key=lambda x: x[0])[0]
    # max_min = mmax - mmin
    # test_label = list(map(lambda x: [(x[0] - mmin)/max_min], test_label))

    # model
    model = NeuralNetwork(len(train_data[0]), HIDDEN_NUM, 1)

    # rst = [model.feed_forward(i)[0] for i in test_data]
    # print(rst)

    # train
    epoch = []
    total_error = []
    cnt = []
    for i in range(EPOCH):
        for j in range(len(train)):
            model.train(train_data[j], train_label[j])
        if i % 10 == 0:
            epoch.append(i)
            total_error.append(round(model.calculate_total_error([[
                train_data[k], train_label[k]] for k in range(len(train))]),
                9)) # [[[0.05, 0.1], [0.01, 0.99]]]
            # test
            rst = [round(model.feed_forward(i)[0], 0) for i in test_data]
            count = 0
            for ind, k in enumerate(rst):
                if (k == test_label[ind][0]):
                    count += 1
            cnt.append(count / len(test_data))
            print("EPOCH:",i)
            print("total_error:", total_error[-1])
            print("Accuracy:", count / len(test_data))

```

```
        plt.plot(epoch, total_error)
        plt.show()
        plt.plot(epoch, cnt)
        plt.show()


if __name__ == '__main__':
    main()
```

- 结果如下图，分别是损失函数的变化和分类成功率的变化。
- 代码在原代码的测例上表现良好。
- 但在该分类问题上效果很差，不管是调整隐藏层节点数，学习率，激活函数，训练次数等超参数，还是进行不同的数据的规范化，设置学习率衰减，进行不同的数据初始化，该问题都没有办法收敛。代价函数仅在前几次训练有一定的变化，后面的几乎没有变化。
- 仔细观察代码运行情况，发现是权重更新太小了，一般只有 $10^{-5} - 10^{-9}$ 数量级的变化，这对 $10^{-1}$ 数量级左右的权重来说基本没有变化。但权重变化是严格按照公式计算的，进一步查看，又发现是当分类输出接近 1 时，sigmod 函数的导数值就接近 0，这使得权重几乎不更新，换句话说 1 是一个局部最优值，它影响了程序的收敛情况。但是没有找到好的解决办法。


128.56
128.55
128.54
128.53
128.52
128.51
128.50
0
20
40
60
80


0.72
0.71
0.70
0.69
0.68
0.67
0.66
0
20
40
60
80