

# E14 BP Algorithm (C++/Python)

---

20214966 Yangkai Lin 20214810 Suixin Ou

2020 年 12 月 11 日

## 目录

<b>1</b>	<b>Horse Colic Data Set</b>	<b>2</b>
<b>2</b>	<b>Reference Materials</b>	<b>2</b>
<b>3</b>	<b>Tasks</b>	<b>7</b>
<b>4</b>	<b>Codes and Results</b>	<b>7</b>

# 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](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

```

1 # -*- coding: utf-8 -*-
2 import random
3 import math
4
5 # Shorthand:
6 # "pd_" as a variable prefix means "partial derivative"
7 # "d_" as a variable prefix means "derivative"
8 # "_wrt_" is shorthand for "with respect to"

```

```

9  # "w_ho" and "w_ih" are the index of weights from hidden to output layer neurons
   and input to hidden layer neurons respectively
10
11 class NeuralNetwork:
12     LEARNING_RATE = 0.5
13     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):
14         #Your Code Here
15
16     def init_weights_from_inputs_to_hidden_layer_neurons(self, hidden_layer_weights
        ):
17         #Your Code Here
18
19     def init_weights_from_hidden_layer_neurons_to_output_layer_neurons(self,
        output_layer_weights):
20         #Your Code Here
21
22     def inspect(self):
23         print('_____')
24         print('*_Inputs:_{ }'.format(self.num_inputs))
25         print('_____')
26         print('Hidden_Layer')
27         self.hidden_layer.inspect()
28         print('_____')
29         print('*_Output_Layer')
30         self.output_layer.inspect()
31         print('_____')
32
33     def feed_forward(self, inputs):
34         #Your Code Here
35
36     # Uses online learning, ie updating the weights after each training case
37     def train(self, training_inputs, training_outputs):
38         self.feed_forward(training_inputs)
39
40         # 1. Output neuron deltas
41         #Your Code Here
42         # E / z
43
44         # 2. Hidden neuron deltas

```

```

45     # We need to calculate the derivative of the error with respect to the
        output of each hidden layer neuron
46     #  $dE/dy = \sum E/z * z/y = \sum E/z * w$ 
47     #  $E/z = dE/dy * z/$ 
48     #Your Code Here
49
50     # 3. Update output neuron weights
51     #  $E/w = E/z * z/w$ 
52     #  $\Delta w = * E/w$ 
53     #Your Code Here
54
55     # 4. Update hidden neuron weights
56     #  $E/w = E/z * z/w$ 
57     #  $\Delta w = * E/w$ 
58     #Your Code Here
59
60     def calculate_total_error(self, training_sets):
61         #Your Code Here
62         return total_error
63
64     class NeuronLayer:
65         def __init__(self, num_neurons, bias):
66
67             # Every neuron in a layer shares the same bias
68             self.bias = bias if bias else random.random()
69
70             self.neurons = []
71             for i in range(num_neurons):
72                 self.neurons.append(Neuron(self.bias))
73
74         def inspect(self):
75             print('Neurons:', len(self.neurons))
76             for n in range(len(self.neurons)):
77                 print('  Neuron', n)
78                 for w in range(len(self.neurons[n].weights)):
79                     print('    Weight:', self.neurons[n].weights[w])
80                 print('    Bias:', self.bias)
81
82         def feed_forward(self, inputs):
83             outputs = []
84             for neuron in self.neurons:

```

```

85         outputs.append(neuron.calculate_output(inputs))
86     return outputs
87
88     def get_outputs(self):
89         outputs = []
90         for neuron in self.neurons:
91             outputs.append(neuron.output)
92         return outputs
93
94 class Neuron:
95     def __init__(self, bias):
96         self.bias = bias
97         self.weights = []
98
99     def calculate_output(self, inputs):
100         #Your Code Here
101
102     def calculate_total_net_input(self):
103         #Your Code Here
104
105         # Apply the logistic function to squash the output of the neuron
106         # The result is sometimes referred to as 'net' [2] or 'net' [1]
107     def squash(self, total_net_input):
108         #Your Code Here
109
110         # Determine how much the neuron's total input has to change to move closer to
111         the expected output
112         #
113         # Now that we have the partial derivative of the error with respect to the
114         output (E/y) and
115         # the derivative of the output with respect to the total net input (dy/dz) we
116         can calculate
117         # the partial derivative of the error with respect to the total net input.
118         # This value is also known as the delta () [1]
119         # = E/z = E/y * dy/dz
120         #
121     def calculate_pd_error_wrt_total_net_input(self, target_output):
122         #Your Code Here
123
124         # The error for each neuron is calculated by the Mean Square Error method:
125     def calculate_error(self, target_output):

```

```

123  #Your Code Here
124
125  # The partial derivate of the error with respect to actual output then is
      calculated by:
126   $\# = 2 * 0.5 * (target\ output - actual\ output) ^ (2 - 1) * -1$ 
127   $\# = -(target\ output - actual\ output)$ 
128  #
129  # The Wikipedia article on backpropagation [1] simplifies to the following, but
      most other learning material does not [2]
130   $\# = actual\ output - target\ output$ 
131  #
132  # Alternative, you can use (target - output), but then need to add it during
      backpropagation [3]
133  #
134  # Note that the actual output of the output neuron is often written as y and
      target output as t so:
135   $\# = E/y = -(t - y)$ 
136  def calculate_pd_error_wrt_output(self, target_output):
137  #Your Code Here
138
139  # The total net input into the neuron is squashed using logistic function to
      calculate the neuron's output:
140   $\# y = 1 / (1 + e^{(-z)})$ 
141  # Note that where represents the output of the neurons in whatever layer we'
      re looking at and represents the layer below it
142  #
143  # The derivative (not partial derivative since there is only one variable) of
      the output then is:
144   $\# dy/dz = y * (1 - y)$ 
145  def calculate_pd_total_net_input_wrt_input(self):
146  #Your Code Here
147
148  # The total net input is the weighted sum of all the inputs to the neuron and
      their respective weights:
149   $\# = z = net = xw + xw \dots$ 
150  #
151  # The partial derivative of the total net input with respective to a given
      weight (with everything else held constant) then is:
152   $\# = z/w = some\ constant + 1 * xw^{(1-0)} + some\ constant \dots = x$ 
153  def calculate_pd_total_net_input_wrt_weight(self, index):
154  #Your Code Here

```

```

155
156 # An example:
157
158 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)
159 for i in range(10000):
160     nn.train([0.05, 0.1], [0.01, 0.99])
161     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

```

1  # -*- coding: utf-8 -*-
2  import random
3  import math
4  import numpy as np
5  import matplotlib.pyplot as plt
6
7  EPOCH = 100
8  HIDDEN_NUM = 20
9  FUN = 0
10
11 def F(x):
12     if FUN == 0:
13         return sigmoid(x)
14     elif FUN == 1:

```

```

15         return relu(x)
16     elif FUN == 2:
17         return tanh(x)
18
19 def d_F(x):
20     if FUN == 0:
21         return d_sigmoid(x)
22     elif FUN == 1:
23         return d_relu(x)
24     elif FUN == 2:
25         return d_tanh(x)
26
27 def relu(x):
28     return max(x, 0)
29
30 def d_relu(x):
31     return 1 if x > 0 else 0
32
33 def sigmoid(x):
34     return 1 / (1 + math.exp(-x))
35
36 def d_sigmoid(x):
37     return x * (1 - x)
38
39 def tanh(x):
40     return math.tanh(x)
41
42 def d_tanh(x):
43     return -x ** 2
44
45 # Shorthand:
46 # "pd_" as a variable prefix means "partial derivative"
47 # "d_" as a variable prefix means "derivative"
48 # "_wrt_" is shorthand for "with respect to"
49 # "w_ho" and "w_ih" are the index of weights from hidden to output layer
50     neurons and input to hidden layer neurons respectively

```



```

51 class NeuralNetwork:
52     LEARNING_RATE = 0.5
53
54     # 这里默认一层隐藏层
55     def __init__(self, num_inputs, num_hidden, num_outputs,
56                 hidden_layer_weights=None, hidden_layer_bias=None,
57                 output_layer_weights=None, output_layer_bias=None):
58         # Your Code Here
59         self.num_inputs = num_inputs
60         self.hidden_layer = NeuronLayer(num_hidden, hidden_layer_bias)
61         self.output_layer = NeuronLayer(num_outputs, output_layer_bias)
62         self.init_weights_from_inputs_to_hidden_layer_neurons(
63             hidden_layer_weights)
64         self.init_weights_from_hidden_layer_neurons_to_output_layer_neurons(
65             output_layer_weights)
66
67     def init_weights_from_inputs_to_hidden_layer_neurons(self,
68                 hidden_layer_weights):
69         # Your Code Here
70         if hidden_layer_weights is not None:
71             for i in range(len(self.hidden_layer.neurons)):
72                 for j in range(self.num_inputs):
73                     self.hidden_layer.neurons[i].weights.append(
74                         hidden_layer_weights[i * self.num_inputs + j])
75         else:
76             for i in range(len(self.hidden_layer.neurons)):
77                 for j in range(self.num_inputs):
78                     self.hidden_layer.neurons[i].weights.append(random.
79                         random())
80                 # self.hidden_layer.neurons[i].weights.append(.5)
81
82     def init_weights_from_hidden_layer_neurons_to_output_layer_neurons(self,
83                 output_layer_weights):
84         # Your Code Here
85         if output_layer_weights is not None:
86             for i in range(len(self.output_layer.neurons)):

```

```

81         for j in range(len(self.hidden_layer.neurons)):
82             self.output_layer.neurons[i].weights.append(
83                 output_layer_weights[i * len(self.hidden_layer.
84                     neurons) + j])
85
86     else:
87         for i in range(len(self.output_layer.neurons)):
88             for j in range(len(self.hidden_layer.neurons)):
89                 self.output_layer.neurons[i].weights.append(random.
90                     random())
91                 # self.output_layer.neurons[i].weights.append(.5)
92
93     def inspect(self):
94         print('-----')
95         print('*_Inputs:{}'.format(self.num_inputs))
96         print('-----')
97         print('Hidden_Layer')
98         self.hidden_layer.inspect()
99         print('-----')
100        print('*_Output_Layer')
101        self.output_layer.inspect()
102        print('-----')
103
104    def feed_forward(self, inputs):
105        # Your Code Here
106        return self.output_layer.feed_forward(self.hidden_layer.feed_forward
107            (inputs))
108
109    # Uses online learning, ie updating the weights after each training case
110    def train(self, training_inputs, training_outputs):
111        model_outputs = self.feed_forward(training_inputs)
112
113        # 1. Output neuron deltas
114        # E / z
115        # Your Code Here
116        output_delta = []
117        for ind, i in enumerate(self.output_layer.neurons):

```

```

113         output_delta.append(i.calculate_pd_error_wrt_total_net_input(
            training_outputs[ind]))
114
115     # 2. Hidden neuron deltas
116     # We need to calculate the derivative of the error with respect to
        the output of each hidden layer neuron
117     #  $dE/dy = \sum E/z * z/y = \sum E/z * w$ 
118     #  $E/z = dE/dy * z/$ 
119     # Your Code Here
120     hidden_delta = []
121     for ind1, i in enumerate(self.hidden_layer.neurons):
122         sum = 0
123         for ind2, j in enumerate(self.output_layer.neurons):
124             sum += j.weights[ind1] * output_delta[ind2]
125         hidden_delta.append(sum * d_F(self.hidden_layer.get_outputs()[
            ind1]))
126
127     # 3. Update output neuron weights
128     #  $E/w = E/z * z/w$ 
129     #  $\Delta w = * E/w$ 
130     # Your Code Here
131     for i in range(len(self.output_layer.neurons)):
132         for j in range(len(self.output_layer.neurons[i].weights)):
133             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]
134
135     # 4. Update hidden neuron weights
136     #  $E/w = E/z * z/w$ 
137     #  $\Delta w = * E/w$ 
138     # Your Code Here
139     for i in range(len(self.hidden_layer.neurons)):
140         for j in range(len(self.hidden_layer.neurons[i].weights)):
141             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]
142         # self.LEARNING_RATE *= 0.9999
143
144     def calculate_total_error(self, training_sets):
145         # Your Code Here
146         total_error = 0
147         for inputs, outputs in training_sets:
148             self.feed_forward(inputs)
149             for ind in range(len(outputs)):
150                 total_error += self.output_layer.neurons[ind].
151                     calculate_error(outputs[ind])
152
153         return total_error
154
155 class NeuronLayer:
156     def __init__(self, num_neurons, bias):
157
158         # Every neuron in a layer shares the same bias
159         self.bias = bias if bias else random.random()
160         # self.bias = bias if bias else 0
161
162         self.neurons = []
163         for i in range(num_neurons):
164             self.neurons.append(Neuron(self.bias))
165
166     def inspect(self):
167         print('Neurons:', len(self.neurons))
168         for n in range(len(self.neurons)):
169             print('  Neuron', n)
170             for w in range(len(self.neurons[n].weights)):
171                 print('    Weight:', self.neurons[n].weights[w])
172             print('    Bias:', self.bias)
173
174     def feed_forward(self, inputs):
175         outputs = []
176         for neuron in self.neurons:
177             outputs.append(neuron.calculate_output(inputs))

```

```

177         return outputs
178
179     def get_outputs(self):
180         outputs = []
181         for neuron in self.neurons:
182             outputs.append(neuron.outputs)
183         return outputs
184
185
186 class Neuron:
187     def __init__(self, bias):
188         self.bias = bias
189         self.weights = []
190
191     def calculate_output(self, inputs):
192         # Your Code Here
193         self.inputs = inputs
194         self.outputs = self.squash(self.calculate_total_net_input())
195         return self.outputs
196
197     def calculate_total_net_input(self):
198         # Your Code Here
199         sum = 0
200         for i in range(len(self.inputs)):
201             sum += self.inputs[i] * self.weights[i]
202         return sum + self.bias # +b
203
204     # Apply the logistic function to squash the output of the neuron
205     # The result is sometimes referred to as 'net' [2] or 'net' [1]
206     def squash(self, total_net_input):
207         # Your Code Here
208         return F(total_net_input)
209
210     # Determine how much the neuron's total input has to change to move
211     # closer to the expected output
212     #

```

```

212     # Now that we have the partial derivative of the error with respect to
        the output ( $E/y$ ) and
213     # the derivative of the output with respect to the total net input ( $dy/dz$ ) we can calculate
214     # the partial derivative of the error with respect to the total net
        input.
215     # This value is also known as the delta () [1]
216     #  $\delta = E/z = E/y * dy/dz$ 
217     #
218     def calculate_pd_error_wrt_total_net_input(self, target_output):
219         # Your Code Here
220         return self.calculate_pd_error_wrt_output(target_output) * self.
            calculate_pd_total_net_input_wrt_input()
221
222     # The error for each neuron is calculated by the Mean Square Error
        method:
223     def calculate_error(self, target_output):
224         # Your Code Here
225         # 均方误差
226         return 0.5 * (target_output - self.outputs) ** 2
227
228     # The partial derivate of the error with respect to actual output then
        is calculated by:
229     #  $= 2 * 0.5 * (target\ output - actual\ output) ^ (2 - 1) * -1$ 
230     #  $= -(target\ output - actual\ output)$ 
231     #
232     # The Wikipedia article on backpropagation [1] simplifies to the
        following, but most other learning material does not [2]
233     #  $= actual\ output - target\ output$ 
234     #
235     # Alternative, you can use  $(target - output)$ , but then need to add it
        during backpropagation [3]
236     #
237     # Note that the actual output of the output neuron is often written as  $y$ 
        and target output as  $t$  so:
238     #  $= E/y = -(t - y)$ 
239     def calculate_pd_error_wrt_output(self, target_output):

```

```

240         # Your Code Here
241         # 均方误差
242         return target_output - self.outputs
243
244     # The total net input into the neuron is squashed using logistic
245     # function to calculate the neuron's output:
246     #  $y = 1 / (1 + e^{-z})$ 
247     # Note that where  $y$  represents the output of the neurons in whatever
248     # layer we're looking at and  $z$  represents the layer below it
249     #
250     # The derivative (not partial derivative since there is only one
251     # variable) of the output then is:
252     #  $dy/dz = y * (1 - y)$ 
253     def calculate_pd_total_net_input_wrt_input(self):
254         # Your Code Here
255         return d_F(self.outputs)
256
257     # The total net input is the weighted sum of all the inputs to the
258     # neuron and their respective weights:
259     #  $z = \sum w_i x_i$ 
260     #
261     # The partial derivative of the total net input with respect to a
262     # given weight (with everything else held constant) then is:
263     #  $\partial z / \partial w_i = x_i$ 
264     def calculate_pd_total_net_input_wrt_weight(self, index):
265         # Your Code Here
266         return self.inputs[index]
267
268     def load(file_name):
269         with open(file_name) as f:
270             rst = []
271             for i in f.readlines():
272                 curr = []
273                 for j in i[:-1].split('␣'):
274                     if j != '':
275                         curr.append(j)

```

```

272         rst.append(curr)
273     return rst
274
275
276 def main():
277     # An example:
278     nn = NeuralNetwork(2, 2, 2, hidden_layer_weights=[0.15, 0.2, 0.25, 0.3],
279                        hidden_layer_bias=0.35,
280                        output_layer_weights=[0.4, 0.45, 0.5, 0.55],
281                        output_layer_bias=0.6)
282
283     for i in range(10):
284         nn.train([0.05, 0.1], [0.01, 0.99])
285         print(i, round(nn.calculate_total_error([[0.05, 0.1], [0.01,
286         0.99]]), 9))
287
288     # load
289     train = load("horse-colic.data")
290     test = load("horse-colic.test")
291
292     # wash
293     # 改未知数据为 -1.0
294     train_data = [i[:22] + i[23:-3] for i in train]
295     train_label = [[float(i[22])] if i[22] != '?' else [-1.0] for i in train
296     ]
297     test_data = [i[:22] + i[23:-3] for i in test]
298     test_label = [[float(i[22])] if i[22] != '?' else [-1.0] for i in test]
299     for i in range(len(train_data)):
300         for j in range(len(train_data[i])):
301             train_data[i][j] = float(train_data[i][j]) if train_data[i][j]
302             != '?' else -1.0
303     for i in range(len(test_data)):
304         for j in range(len(test_data[i])):
305             test_data[i][j] = float(test_data[i][j]) if test_data[i][j]
306             != '?' else -1.0
307
308     # 属性归一化
309     for i in range(len(train_data[0])):

```



```

303     mmax = max([k[i] for k in train_data])
304     mmin = min([k[i] for k in train_data])
305     max_min = mmax - mmin
306     # j = max([math.log10(k[i]) if k[i]>0 else 0 for k in train_data])
307     # mmean = np.mean([k[i] for k in train_data])
308     # sstd = np.std([k[i] for k in train_data])
309     for j in range(len(train_data)):
310         # 最小最大规范化-
311         # train_data[j][i] = (train_data[j][i] - mmin)/(max_min if
312             max_min > 0 else (mmax if mmax else 1))
313         train_data[j][i] = (train_data[j][i] - mmin)/max_min
314         # 小数定标规范化
315         # train_data[j][i] /= 10 ** j
316         # 零均值规范化-
317         # train_data[j][i] = (train_data[j][i] - mmean)/sstd
318
319     for i in range(len(test_data[0])): #
320         mmax = max([k[i] for k in test_data])
321         mmin = min([k[i] for k in test_data])
322         max_min = mmax - mmin
323         # j = max([math.log10(k[i]) if k[i]>0 else 0 for k in test_data])
324         # mmean = np.mean([k[i] for k in test_data])
325         # sstd = np.std([k[i] for k in test_data])
326         for j in range(len(test_data)):
327             # 最小最大规范化-
328             # test_data[j][i] = (test_data[j][i] - mmin)/(max_min if max_min
329                 > 0 else (mmax if mmax else 1))
330             test_data[j][i] = (test_data[j][i] - mmin)/max_min
331             # 小数定标规范化
332             # test_data[j][i] /= 10 ** j
333             # 零均值规范化-
334             # test_data[j][i] = (test_data[j][i] - mmean) / sstd
335
336     # # 标签归一化
337     # mmax = max(train_label, key=lambda x: x[0])[0]
338     # mmin = min(train_label, key=lambda x: x[0])[0]
339     # max_min = mmax - mmin

```

```

338     # train_label = list(map(lambda x: [(x[0] - mmin)/max_min], train_label)
339     )
340     #
341     # mmax = max(test_label, key=lambda x: x[0])[0]
342     # mmin = min(test_label, key=lambda x: x[0])[0]
343     # max_min = mmax - mmin
344     # test_label = list(map(lambda x: [(x[0] - mmin)/max_min], test_label))
345
346     # model
347     model = NeuralNetwork(len(train_data[0]), HIDDEN_NUM, 1)
348
349     # rst = [model.feed_forward(i)[0] for i in test_data]
350     # print(rst)
351
352     # train
353     epoch = []
354     total_error = []
355     cnt = []
356     for i in range(EPOCH):
357         for j in range(len(train)):
358             model.train(train_data[j], train_label[j])
359         if i % 10 == 0:
360             epoch.append(i)
361             total_error.append(round(model.calculate_total_error([[
362                 train_data[k], train_label[k] for k in range(len(train))
363             ]], 9)) # [[0.05, 0.1], [0.01, 0.99]])
364
365     # test
366     rst = [round(model.feed_forward(i)[0], 0) for i in test_data]
367     count = 0
368     for ind, k in enumerate(rst):
369         if (k == test_label[ind][0]):
370             count += 1
371     cnt.append(count / len(test_data))
372     print("EPOCH:", i)
373     print("total_error:", total_error[-1])
374     print("Accuracy:", count / len(test_data))

```

```
372     plt.plot(epoch, total_error)
373     plt.show()
374     plt.plot(epoch, cnt)
375     plt.show()
376
377
378 if __name__ == '__main__':
379     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，sigmoid 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。

