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

E14 BP Algorithm (C++/Python)

17341068 Yangfan Jiang

December 15, 2019

Contents

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

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

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

# 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 = 0.05
    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.num_hidden = num_hidden
        self.num_outputs = num_outputs
        self.hidden_layer_weights = hidden_layer_weights
        self.hidden_layer_bias = hidden_layer_bias
        self.output_layer_weights = output_layer_weights
        self.output_layer_bias = output_layer_bias

        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 not hidden_layer_weights:
            self.hidden_layer_weights = [random.random() for i in range(self.num_inputs
                * self.num_hidden)]
```

```python
        begin = 0
        for neuron in self.hidden_layer.neurons:
            neuron.weights = self.hidden_layer_weights[begin : begin+self.num_inputs]
            begin += self.num_inputs


    def init_weights_from_hidden_layer_neurons_to_output_layer_neurons(self, output_layer_weights):
        #Your Code Here
        if not output_layer_weights:
            self.output_layer_weights = [random.random() for i in range(self.num_hidden * self.num_outputs)]
        begin = 0
        for neuron in self.output_layer.neurons:
            neuron.weights = self.output_layer_weights[begin : begin+self.num_hidden]
            begin += self.num_hidden

    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
        self.hidden_layer_outputs = self.hidden_layer.feed_forward(inputs)
        self.output_layer_outputs = self.output_layer.feed_forward(self.hidden_layer_outputs)
        return self.output_layer_outputs

    # 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
        # E/z
```

```
        # Your Code Here
        output_neuron_deltas = []
        for j, neuron in enumerate(self.output_layer.neurons):
            output_neuron_deltas.append(neuron.calculate_pd_error_wrt_total_net_input(
                training_outputs[j]))

        # 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_neuron_deltas = []
        for h, neuron_h in enumerate(self.hidden_layer.neurons):
            Sum = 0
            for j, neuron_o in enumerate(self.output_layer.neurons):
                Sum += neuron_o.weights[h] * output_neuron_deltas[j]
            tmp = neuron_h.output * (1 - neuron_h.output)
            hidden_neuron_deltas.append(tmp * Sum)

        # 3. Update output neuron weights
        # E/w = E/z * z/w
        # Δw = * E/w
        # Your Code Here
        for j, neuron_o in enumerate(self.output_layer.neurons):
            for h, neuron_h in enumerate(self.hidden_layer.neurons):
                neuron_o.weights[h] += self.LEARNING_RATE * output_neuron_deltas[j] *
                    neuron_h.output

        # 4. Update hidden neuron weights
        # E/w = E/z * z/w
        # Δw = * E/w
        # Your Code Here
        for h, neuron_h in enumerate(self.hidden_layer.neurons):
            for i, x in enumerate(training_inputs):
                neuron_h.weights[i] += self.LEARNING_RATE * hidden_neuron_deltas[h] * x

        # 5. update output layer bias
        # for j, neuron in enumerate(self.output_layer.neurons):
        #     neuron.bias += -1 * self.LEARNING_RATE * output_neuron_deltas[j]

```

```
        # # 6. update hidden layer bias
        # for h, neuron in enumerate(self.hidden_layer.neurons):
        #     neuron.bias += -1 * self.LEARNING_RATE * hidden_neuron_deltas[h]

    def calculate_total_error(self, training_sets):
        #Your Code Here
        total_error = 0
        for case in training_sets:
            training_inputs, training_outputs = case
            self.feed_forward(training_inputs)
            for i, target_output in enumerate(training_outputs):
                total_error += self.output_layer.neurons[i].calculate_error(
                    target_output)
        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):
        self.inputs = inputs

        # Calculate total new input
        total_net_input = 0
        for i in range(len(self.weights)):
            total_net_input += inputs[i] * self.weights[i]

        # Apply the logistic function to squash the output of the neuron
        # Here we use sigmoid fucntion
        x = total_net_input - self.bias
        self.output = 1 / (1 + math.e**(-x))

        return self.output

    # 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
        a = self.calculate_pd_error_wrt_output(target_output)
        b = self.calculate_pd_total_net_input_wrt_input()
        return -1 * a * b
```

```python

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

    # 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 self.output * (1 - self.output)

    # 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 get_training_sets(filename):
    training_sets = []
    data = pd.read_csv(filename)

    for line in data.values:
        attr = list(line)
        target = attr.pop(22)
        training_output = [0.90 if i == (target-1) else 0.05 for i in range(3)]
        training_input = attr
        training_sets.append([training_input, training_output])
    return training_sets


training_sets = get_training_sets('horse-colic-data.csv')
nn = NeuralNetwork(len(training_sets[0][0]), 3, len(training_sets[0][1]))
total, correct = 0, 0
epoch = 10
decay = 0.99
x = []
y = []
for i in range(epoch*300):
    #x.append(i)
    training_inputs, training_outputs = training_sets[i%len(training_sets)]
    nn.train(training_inputs, training_outputs)
    #y.append(nn.calculate_total_error(training_sets))
    if i % 300 == 0:
        print('epoch:', i // 300, 'err:', nn.calculate_total_error(training_sets))
        nn.LEARNING_RATE *= decay
    if i >= (epoch-1)*300:
        output_idx = nn.output_layer_outputs.index(max(nn.output_layer_outputs))
        target_idx = training_outputs.index(max(training_outputs))
        total += 1
```

```
        correct += output_idx == target_idx
print("Accuracy on training set:\n" ,correct/total)

#plt.xlabel("train data")
#plt.ylabel("Error")
#plt.title('bp algorithm')
#plt.plot(x, y, linewidth = 2, label = 'error')
#plt.show()


testing_sets = get_training_sets('horse-colic-test.csv')
total, correct = 0, 0
for i, test_case in enumerate(testing_sets):
    input, target = test_case
    output = nn.feed_forward(input)
    target_idx = target.index(max(target))
    output_idx = output.index(max(output))
    total += 1
    correct += target_idx == output_idx
print("Accuracy on testing set:\n" ,correct/total)
```

Results

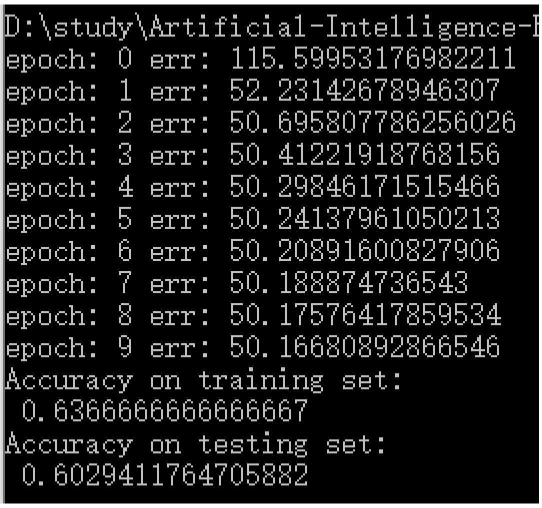

Figure 1: Result

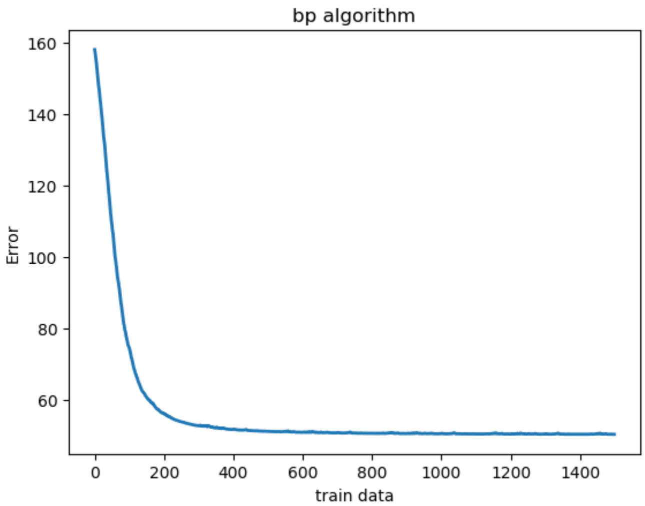

Figure 2: Training curve