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

E14 BP Algorithm (C++/Python)

18340149 孙新梦

2020 年 12 月 10 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 = α *  E / w
        #Your Code Here

        # 4. Update hidden neuron weights
        #  E / w =  E / z *  z / w
```

```
        # Δw = α *   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 δ() [1]
```

```
    # δ =  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 = Φ = 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

code

```
"""孙新梦

    18340149
    BP.py
"""
import numpy as np
import copy
from math import exp, tanh, sqrt
from collections import Counter
import matplotlib.pyplot as plt
import random

# 激活函数
activate_functions = {
    'sigmoid': lambda x: 1 / (1 + exp(-x)),
    'tanh': lambda x: tanh(x),
    'relu': lambda x: 0 if x <= 0 else x,
    'softmax': lambda x, exp_sum_except_x: exp(x) / (
        exp_sum_except_x + exp(x)),
```

```
    'linear': lambda x: x
}

# 激活函数的导函数
activate_function_derivatives = {
    'sigmoid': lambda x: exp(-x) / ((1 + exp(-x)) ** 2),
    'tanh': lambda x: 1 - tanh(x) ** 2,
    'relu': lambda x: 0 if x <= 0 else 1,
    'softmax': lambda x, exp_sum_except_x: exp_sum_except_x * exp(x) / ((exp_sum_except_x + exp(x)) ** 2),
    'linear': lambda x: 1
}


class NeuronNetwork(object):
    # 初始化神经网络:
    def __init__(self, layer_dict, weight_init_dict, bias_init_dict, activate_type_ = 'relu'):
        self.neuron_layer = {}
        if 'output' in layer_dict:
            self.derivative = np.zeros((layer_dict['output'][0], 1))
        if 'softmax' in layer_dict:
            self.derivative = np.zeros((layer_dict['softmax'][0], 1))
        for layer_type_, layer_neurons in layer_dict.items():
            self.neuron_layer[layer_type_] = []
            for i, num in enumerate(layer_neurons):
                self.neuron_layer[layer_type_].append(
                    NeuronLayer(num, bias_init_dict[layer_type_][i], activate_type = activate_type_, layer_type = layer_type_)
                )
```

```
                if layer_type_ != 'input':
                    self.neuron_layer[layer_type_][i].setInputWeight(weight_init_dict[layer_type_][i])

    # 设置超参数
    def setHyperparameters(self, learning_rate = 1e-3, alpha = 0.9, beta = 0.9, epochs = 300):
        self.learning_rate = learning_rate
        self.alpha = alpha
        self.beta = beta
        self.epochs = epochs
        self.loss_epochs = [0 for _ in range(epochs)]
        self.accuracy_epochs = [0 for _ in range(epochs)]

    # 设置验证集
    def setValidationData(self, val_features, val_labels):
        self.val_features = val_features
        self.val_labels = val_labels
        self.val_accuracy_epochs = [0 for _ in range(epochs)]

    # 设置优化器
    def setOptimizer(self, optimizer_type = 'simple'):
        self.optimizer_type = optimizer_type

    def setResultDict(self, res_list):
        self.res_list = res_list

    # 扩展一层的输出，方便下一层的输入
    def expandOutput(self, output, n):
        if output.shape[0] == 1:
            expanded_output = np.zeros((output.shape[1], n))
            for i in range(n):
```

```
                expanded_output[:, i] = output[0, :]
        else:
            expanded_output = np.zeros((output.shape[0], n))
            for i in range(n):
                expanded_output[:, i] = output[:, 0]
        return expanded_output

    # 预测一条数据
    def predictOne(self, feature, label, is_training = True, is_detailed = False):
        # 计算输入层输出
        self.neuron_layer['input'][0].setInput(feature)
        hidden_output = self.neuron_layer['input'][0].calOutput()

        # 计算隐藏层输出
        for i, x in enumerate(self.neuron_layer['hidden']):
            expanded_output = self.expandOutput(hidden_output, len(x.neurons))
            self.neuron_layer['hidden'][i].setInput(expanded_output)
            hidden_output = self.neuron_layer['hidden'][i].calOutput()

        # 计算输出层输出
        if 'output' in self.neuron_layer:
            expanded_output = self.expandOutput(hidden_output, len(self.neuron_layer['output'][0].neurons))
            self.neuron_layer['output'][0].setInput(expanded_output)
            hidden_output = self.neuron_layer['output'][0].calOutput()

        # 计算层输出softmax
```

```
        if 'softmax' in self.neuron_layer:
            expanded_output = self.expandOutput(hidden_output, len(self.neuron_layer['softmax'][0].neurons))
            self.neuron_layer['softmax'][0].setInput(expanded_output)
            hidden_output = self.neuron_layer['softmax'][0].calOutput()

        # 对于每一个样例，E = (output - label) ^ 2，计算dE / d(output)
        self.derivative += hidden_output - label
        index = np.argmax(hidden_output, axis = 0)
        predict = self.res_list[index[0]]
        index = np.argmax(label, axis = 0)
        true_res = self.res_list[index[0]]

        # 比较是否正确
        flag = (predict == true_res)
        if not is_training:
            if is_detailed:
                print预测(":", predict, end = " ")
                print结果(":", true_res, end = " ")
                if flag:
                    print("True")
                else:
                    print("False")

            return 0, flag

        if 'softmax' in self.neuron_layer:
            return self.neuron_layer['softmax'][0].calLoss(label), flag
        return self.neuron_layer['output'][0].calLoss(label), flag
```

```python

    # 预测全部输入数据
    def predictAll(self, features, labels, is_training = True, is_detailed = False):
        assert(features.shape[0] == labels.shape[0])
        feature_len = features.shape[1]
        label_len = labels.shape[1]

        total_loss, total_accuracy = 0, 0
        for i in range(features.shape[0]):
            now_loss, now_accuracy = self.predictOne(features[i].reshape((1, feature_len)), labels[i].reshape((label_len, 1)), is_training, is_detailed)
            total_loss += now_loss
            total_accuracy += now_accuracy

        total_loss /= features.shape[0]
        total_accuracy /= features.shape[0]
        # dE / d(output) = 1 / n * sum(output - label)
        self.derivative /= features.shape[0]

        return total_loss, total_accuracy

    # 误差逆传播
    def backPropagation(self, iter_num = None):
        forward_derivative = self.derivative

        if self.optimizer_type == 'simple':
            if 'softmax' in self.neuron_layer:
                forward_derivative = self.neuron_layer['softmax'][0].simpleBackwardUpdate(forward_derivative, self.learning_rate)
            if 'output' in self.neuron_layer:
```

```
                forward_derivative = self.neuron_layer['output
                    '][0].simpleBackwardUpdate(forward_derivative,
                    self.learning_rate)

            i = len(self.neuron_layer['hidden']) - 1
            while i >= 0:
                forward_derivative = self.neuron_layer['hidden'][i
                    ].simpleBackwardUpdate(forward_derivative, self
                    .learning_rate)
                i -= 1
        else:
            if 'softmax' in self.neuron_layer:
                forward_derivative = self.neuron_layer['softmax
                    '][0].adamBackwardUpdate(forward_derivative,
                    iter_num,
                                                                                       self
                                                                                           .
                                                                                           learning
                                                                                           ,
                                                                                           self
                                                                                           .
                                                                                           alpha
                                                                                           ,
                                                                                           self
                                                                                           .
                                                                                           beta
                                                                                           )

            if 'output' in self.neuron_layer:
                forward_derivative = self.neuron_layer['output
                    '][0].adamBackwardUpdate(forward_derivative,
```

```
                        iter_num,
                                                                                 self
                                                                                   .
                                                                                   learning
                                                                                   ,
                                                                                   self
                                                                                   .
                                                                                   alpha
                                                                                   ,
                                                                                   self
                                                                                   .
                                                                                   beta
                                                                                   )

            i = len(self.neuron_layer['hidden']) - 1
            while i >= 0:
                forward_derivative = self.neuron_layer['hidden'][i
                    ].adamBackwardUpdate(forward_derivative,
                    iter_num,
                                                                               self
                                                                               .
                                                                               learning_rat
                                                                               ,
                                                                               self
                                                                               .
                                                                               alpha
                                                                               ,
                                                                               self
```

```
.
beta
)

                    i -= 1

        # 训练
        def train(self, train_features, train_labels):
            best_accuracy = 0
            best_epoch = 0
            for i in range(self.epochs):
                if i > 0 and i % 200 == 0:
                    self.learning_rate /= 2
                self.loss_epochs[i], self.accuracy_epochs[i] = self.
                    predictAll(train_features, train_labels)
                print第(" %d 代, Loss: %.6f"%(i, self.loss_epochs[i]))
                print准确度(": %.6f"%(self.accuracy_epochs[i]))

                self.backPropagation(i)
                _, self.val_accuracy_epochs[i] = self.predictAll(self.
                    val_features, self.val_labels, is_training = False)
                if best_accuracy < self.val_accuracy_epochs[i]:
                    best_accuracy = self.val_accuracy_epochs[i]
                    best_epoch = i
                print("Val Accuracy:", self.val_accuracy_epochs[i])
            print最佳准确
              度(" %.6f 第 %d 代"%(best_accuracy, best_epoch))

        # 生成正确率和的图像loss
        def plotInfo(self):
            x = list(range(0, self.epochs))
            plt.figure(1)

```

```
        plt.xlabel代('')
        plt.ylabel('值loss')

        plt.plot(x, self.loss_epochs, 'b-^', linewidth = 2)
        plt.savefig('loss.png')

        plt.figure(2)
        plt.xlabel('epochs')
        plt.ylabel('accuracy')

        plt.plot(x, self.accuracy_epochs, 'c-x', linewidth = 2)
        plt.plot(x, self.val_accuracy_epochs, 'r-.', linewidth =
            2)
        plt.legend(['train', 'val'])
        plt.savefig('accuracy.png')

    def predict(self, test_features, test_labels):
        _, accuracy = self.predictAll(test_features, test_labels,
            is_training = False, is_detailed = True)
        print("Test Accuracy:", accuracy)

class NeuronLayer(object):
    # 激活函数：层用，层用，隐藏层可自行设定激活函
        数outputsigmoidsoftmaxsoftmax
    def __init__(self, num_neurons, bias, activate_type = 'relu',
        layer_type = 'hidden'):
        self.bias = bias
        self.bias_first_moment = np.zeros(bias.shape)
        self.bias_second_moment = np.zeros(bias.shape)
        self.layer_type = layer_type
        if layer_type == 'input':
            self.neurons = [Neuron(0, 'linear') for i in range(
                num_neurons)]
```

```
        elif layer_type == 'softmax':
            self.neurons = [Neuron(bias[i, 0], 'softmax') for i in range(num_neurons)]
        elif layer_type == 'output':
            self.neurons = [Neuron(bias[i, 0], 'sigmoid') for i in range(num_neurons)]
        else:
            self.neurons = [Neuron(bias[i, 0], activate_type) for i in range(num_neurons)]
        self.input = None
        self.input_weight = None
        self.exp_sum = None
        self.output = np.zeros((num_neurons, 1))

    # inputs: m * n(为上一层神经元数目，为当前层神经元数目mn)
    def setInput(self, inputs):
        assert(inputs.shape[1] == len(self.neurons))
        self.inputs = copy.deepcopy(inputs)
        for i in range(len(self.neurons)):
            self.neurons[i].setInput(inputs[:, i].reshape((inputs.shape[0], 1)))

    # 记录入权重：m * n
    def setInputWeight(self, weights):
        assert(self.layer_type != 'input')
        assert(weights.shape[1] == len(self.neurons))
        self.weights = copy.deepcopy(weights)
        self.weight_first_moment = np.zeros(weights.shape)
        self.weight_second_moment = np.zeros(weights.shape)
        for i in range(len(self.neurons)):
            self.neurons[i].setInputWeight(weights[:, i].reshape((weights.shape[0], 1)))

```

```python
    # 计算输出：n * 1
    def calOutput(self):
        if self.layer_type == 'input':
            self.output = copy.deepcopy(self.inputs)
        else:
            if self.layer_type == 'softmax':
                self.exp_sum = 0

            for i, x in enumerate(self.neurons):
                self.output[i, 0] = x.calOutput()
                # 层计算的和softmaxexp
                if self.layer_type == 'softmax':
                    self.exp_sum += self.output[i, 0]

            if self.layer_type == 'softmax':
                for i, _ in enumerate(self.neurons):
                    self.output[i, 0] /= self.exp_sum
                    self.neurons[i].output /= self.exp_sum
        return self.output

    # 计算损失（只有层或层）softmaxoutput
    def calLoss(self, label):
        assert(self.layer_type == 'softmax' or self.layer_type ==
            'output')
        return 0.5 * (np.linalg.norm(self.output - label, 2) ** 2)

    # 梯度下降（普通的，是上一层传递过来的梯度）GDforward_derivatives
    def simpleBackwardUpdate(self, forward_derivatives,
        learning_rate = 1e-3):
        exp_sum = self.exp_sum
        last_layer_shape = self.weights.shape[0]
        now_derivatives = np.zeros((last_layer_shape, 1))

```

```python
        for i, x in enumerate(self.neurons):
            bias_derivative, weight_derivative, input_derivative = x.calBackwardDerivative(forward_derivatives[i, 0], exp_sum)

            # 层只传播梯度，不更新权重和softmaxbias
            if self.layer_type != 'softmax':
                self.bias[i, 0] -= learning_rate * bias_derivative
                self.weights[:, i] -= learning_rate * weight_derivative[:, 0]
                self.neurons[i].setBias(self.bias[i, 0])
                self.neurons[i].setInputWeight(self.weights[:, i].reshape((last_layer_shape, 1)))

                now_derivatives += input_derivative
            else:
                now_derivatives[i, 0] = bias_derivative

        return now_derivatives

    # 梯度下降（自己实现的）adam
    def adamBackwardUpdate(self, forward_derivatives, iter_num, learning_rate = 1e-3, alpha = 0.9, beta = 0.9):
        exp_sum = self.exp_sum
        last_layer_shape = self.weights.shape[0]
        now_derivatives = np.zeros((last_layer_shape, 1))
        for i, x in enumerate(self.neurons):
            bias_derivative, weight_derivative, input_derivative = x.calBackwardDerivative(forward_derivatives[i, 0], exp_sum)

            # 层只传播梯度，不更新权重和softmaxbias
            if self.layer_type != 'softmax':
```

```
                first_moment = alpha * self.bias_first_moment[i, 0] + (1 - alpha) * bias_derivative
                second_moment = beta * self.bias_second_moment[i, 0] + (1 - beta) * bias_derivative * bias_derivative
                first_moment_unbias = first_moment / (1 - alpha ** (iter_num + 1))
                second_moment_unbias = second_moment / (1 - beta ** (iter_num + 1))
                self.bias[i, 0] -= learning_rate * first_moment_unbias / (np.sqrt(second_moment_unbias) + 1e-7)
                self.bias_first_moment[i, 0] = first_moment
                self.bias_second_moment[i, 0] = second_moment

                weight_first_moment = alpha * self.weight_first_moment[:, i] + (1 - alpha) * weight_derivative[:, 0]
                weight_second_moment = beta * self.weight_second_moment[:, i] + (1 - beta) * (weight_derivative[:, 0] ** 2)
                weight_first_moment_unbias = weight_first_moment / (1 - alpha ** (iter_num + 1))
                weight_second_moment_unbias = weight_second_moment / (1 - beta ** (iter_num + 1))
                self.weights[:, i] -= learning_rate * weight_first_moment_unbias / (np.sqrt(weight_second_moment_unbias) + 1e-7)
                self.weight_first_moment[:, i] = weight_first_moment
                self.weight_second_moment[:, i] = weight_second_moment

```

```
                x.setBias(self.bias[i, 0])
                x.setInputWeight(self.weights[:, i].reshape((last_layer_shape, 1)))
                now_derivatives += input_derivative
            else:
                now_derivatives[i, 0] = bias_derivative

        return now_derivatives

class Neuron(object):
    def __init__(self, bias, activate_type = 'relu'):
        self.bias = bias
        self.input = None
        self.input_weight = None
        self.activate_type = activate_type
        self.activate_function = activate_functions[activate_type]
        self.derivative_function = activate_function_derivatives[activate_type]
        self.output = 0
        self.linear_output = 0

    # 设置神经元输入：m * （1是上一层神经元数量）m
    def setInput(self, inputs):
        self.input = inputs

    # 设置神经元入权重：m * （1是上一层神经元数量）m
    def setInputWeight(self, weight):
        self.input_weight = copy.deepcopy(weight)

    # 设置神经元：biasm * （1是上一层神经元数量）m
    def setBias(self, new_bias):
        self.bias = new_bias

```

```python
    # 计算单个神经元输出
    def calOutput(self, exp_sum = None):
        if self.activate_type == 'linear':
            self.output = self.input
        else:
            # y = Wx + b
            self.linear_output = np.dot(self.input.T, self.
                input_weight) + self.bias
            if self.activate_type == 'softmax':
                self.output = exp(self.linear_output)
            else:
                self.output = self.activate_function(self.
                    linear_output)
        return self.output

    # 每个神经元计算梯度
    def calBackwardDerivative(self, forward_derivative, exp_sum =
        None):
        # 计算df / dy(是激活函数f, 是线性输出y)
        if exp_sum == None:
            function_derivative = self.derivative_function(self.
                linear_output)
        else:
            function_derivative = self.derivative_function(self.
                linear_output, exp_sum - self.output)
        bias_derivative = forward_derivative * function_derivative

        input_len = self.input.shape[0]
        weight_derivative = np.zeros((input_len, 1))
        input_derivative = np.zeros((input_len, 1))
        if self.activate_type != 'softmax':
            for i in range(input_len):
                # df / dw = df / dy * dy / dw
```

```python
                weight_derivative[i, 0] = bias_derivative * self.input[i, 0]
                # df / dx = df / dy * dy / dx
                input_derivative[i, 0] = bias_derivative * self.input_weight[i, 0]

        return bias_derivative, weight_derivative, input_derivative

class DataHandler(object):
    def __init__(self, file_road, linear_indexs, ignore_indexs):
        self.file_road = file_road
        self.linear_indexs = linear_indexs
        self.ignore_indexs = ignore_indexs
        self.features = []
        self.labels = []

    # [3, 4, 5, 15]
    def isLinear(self, feature_index):
        return feature_index in self.linear_indexs

    # [2]
    def isIgnore(self, feature_index):
        return feature_index in self.ignore_indexs

    # 读数据
    def readData(self):
        with open(self.file_road, 'r') as f:
            j = 0
            for line in f.readlines():
                now_line = line.strip('\n').strip(' ').split(' ')
                if now_line[0] == '':
                    continue
```

```
                # 只要前列数据（第列是2323）label
                now_line = now_line[:23]
                if now_line[-1] == '?':
                    self.labels.append(float(1.0))
                else:
                    self.labels.append(float(now_line[-1]))
                self.features.append([])
                for i, x in enumerate(now_line[:-1]):
                    if x != '?':
                        if x[0] != '0':
                            self.features[j].append(float(x))
                        else:
                            self.features[j].append(x)
                    else:
                        self.features[j].append(x)
                j += 1

    # 处理features
    def handleFeature(self):
        valid_feature_len = len(self.features[0]) - len(self.ignore_indexs)
        features = np.zeros((len(self.features), valid_feature_len))
        j = 0
        for i in range(features.shape[1]):
            if self.isIgnore(i):
                continue
            feature_col = [feature[j] for feature in self.features]
            no_miss_feature_col = [x for x in feature_col if x != '?']
            update_num = 0
            if not self.isLinear(i):
```

```
                update_num = 0
            else:
                update_num = sum(no_miss_feature_col) / len(no_miss_feature_col)

            # 对于缺失的数据，若为离散量，补；若为连续量，补均值0
            for k in range(features.shape[0]):
                if feature_col[k] == '?':
                    features[k, j] = update_num
                else:
                    features[k, j] = feature_col[k]
            # 标准化
            features[:, j] = (features[:, j] - np.mean(features[:, j])) / np.std(features[:, j])
            j += 1

        self.features = features

    # 处理label
    def handleLabel(self, classfy = 3):
        counter = Counter(self.labels).most_common()

        res_dict = {}
        for i, x in enumerate(counter):
            res_dict[x[0]] = i

        labels = np.zeros((len(self.labels), classfy))
        for i, label in enumerate(self.labels):
            labels[i, res_dict[label]] += 1

        self.labels = labels
        return res_dict

```

```python
    # 数据增强（最后没用到）
    def argumentData(self):
        data_len = self.features.shape[0]
        total_add_feature = None
        total_add_label = None
        flag = 0
        for i in range(data_len):
            index = np.argmax(self.labels[i], axis = 0)
            add_size = (index + 1)
            add_feature = np.zeros((add_size, self.features.shape
                [1]))
            add_label = np.zeros((add_size, self.labels.shape[1]))

            for j in range(add_size):
                add_feature[j, :] = self.features[i, :]
                add_label[j, :] = self.labels[i, :]

            for j in range(self.features.shape[1]):
                if self.isLinear(j + 1):
                    add_feature[:, j] += 0.0001 * np.random.randn(
                        add_size)

            if flag == 0:
                total_add_feature = add_feature
                total_add_label = add_label
                flag = 1
            else:
                total_add_feature = np.concatenate((
                    total_add_feature, add_feature), axis = 0)
                total_add_label = np.concatenate((total_add_label,
                    add_label), axis = 0)
        self.features = np.concatenate((self.features,
            total_add_feature), axis = 0)
```

```
        self.labels = np.concatenate((self.labels, total_add_label
            ), axis = 0)

    def readAndHandle(self, argument = False, classfy = 3):
        self.readData()
        self.handleFeature()
        self.handleLabel(classfy)
        if argument:
            self.argumentData()

if __name__ == '__main__':
    classfy_ = 3
    print("Read And Handle training data ...")
    train_data_handler = DataHandler('horse-colic.data',
                                     linear_indexs = [3, 4, 5, 15],
                                     ignore_indexs = [2])
    train_data_handler.readAndHandle(classfy = classfy_)
    print("Done ...")

    print("Read And Handle testing data ...")
    test_data_handler = DataHandler('horse-colic.test',
                                     linear_indexs = [3, 4, 5, 15],
                                     ignore_indexs = [2])
    test_data_handler.readAndHandle(classfy = classfy_)
    print("Done ...")

    layer_dict = {
        'input': [21],
        'hidden': [12],
        'output': [3]
        # 'softmax': [3],
    }
    weight_init_dict = {
```

```
        'input': [None],
        'hidden': [np.random.randn(21, 12) / (sqrt(21 * 12) * 0.5)],
        'output': [np.random.randn(12, 3) / sqrt(12 * 3)]
        # 'softmax': [np.eye(3)],
    }
    bias_init_dict = {
        'input': [np.zeros((21, 1))],
        'hidden': [np.zeros((12, 1))],
        'output': [np.zeros((3, 1))]
        # 'softmax': [np.zeros((3, 1))],
    }
    learning_rate = 2e-3

    # 的两个参数adam
    alpha = 0.8
    beta = 0.8
    epochs = 400
    # 隐藏层激活函数类型
    activate = 'relu'
    # 优化器
    optimizer = 'adam'

    my_nn = NeuronNetwork(layer_dict, weight_init_dict,
        bias_init_dict, activate)
    my_nn.setHyperparameters(learning_rate, alpha, beta, epochs)
    my_nn.setOptimizer(optimizer)
    my_nn.setResultDict([1.0, 2.0, 3.0])

    print("——————————————————————")
    print("Training ...")
    train_features = train_data_handler.features
    train_labels = train_data_handler.labels
```

```
    val_features = test_data_handler.features
    val_labels = test_data_handler.labels

    my_nn.setValidationData(val_features, val_labels)
    my_nn.train(train_features, train_labels)
    print("Done ...")

    print("————————————————")
    print("Ploting ...")
    my_nn.plotInfo()
    print("Done ...")

    print("————————————————")
    print("Testing ...")
    my_nn.predict(test_data_handler.features, test_data_handler.
        labels)
    print("Done ...")
```

result 下面三张图分别说明了我们输出的几个部分：一个是训练的时候输出loss，准确度，之后看到经过400代训练，准确度依次提升，可以达到训练集最佳0.72准确度

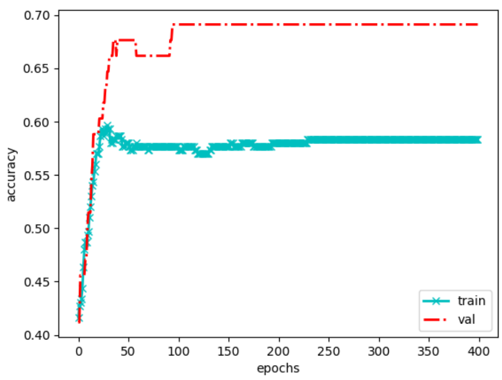
之后测试集上进行测试，看到也有0.72的准确度。这两张是程序运行过程中画出的准确度和Loss值的图像，看到准确度刚开始的训练大幅度提升，之后趋于平缓，loss值则有下降的趋势

```
Training ...
第 0 代, Loss: 0.397882
准确度: 0.170000
Val Accuracy: 0.17647058823529413
第 1 代, Loss: 0.396291
准确度: 0.166667
Val Accuracy: 0.17647058823529413
第 2 代, Loss: 0.394804
准确度: 0.173333
Val Accuracy: 0.17647058823529413
第 3 代, Loss: 0.393394
准确度: 0.186667
Val Accuracy: 0.22058823529411764
第 4 代, Loss: 0.392053
准确度: 0.216667
Val Accuracy: 0.22058823529411764
第 5 代, Loss: 0.390792
准确度: 0.233333
Val Accuracy: 0.25
第 6 代, Loss: 0.389606
准确度: 0.240000
Val Accuracy: 0.2647058823529412
第 7 代, Loss: 0.388490
准确度: 0.256667
Val Accuracy: 0.27941176470588236
```

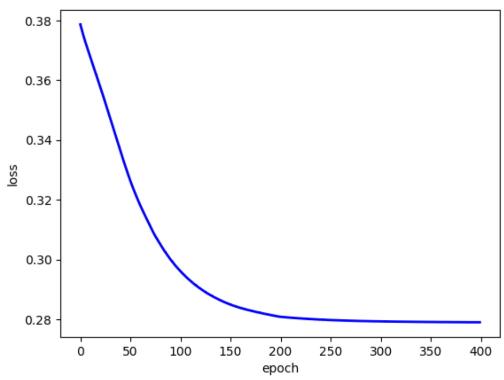
```
Val Accuracy: 0.7205882352941176
第 394 代, Loss: 0.272213
准确度: 0.590000
Val Accuracy: 0.7205882352941176
第 395 代, Loss: 0.272173
准确度: 0.590000
Val Accuracy: 0.7205882352941176
第 396 代, Loss: 0.272141
准确度: 0.590000
Val Accuracy: 0.7205882352941176
第 397 代, Loss: 0.272112
准确度: 0.590000
Val Accuracy: 0.7205882352941176
第 398 代, Loss: 0.272075
准确度: 0.590000
Val Accuracy: 0.7205882352941176
第 399 代, Loss: 0.272040
准确度: 0.590000
Val Accuracy: 0.7205882352941176
最佳准确度 0.720588 第 198 代
Done ...
```

```
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 3.0 False
预测: 1.0 结果: 2.0 False
预测: 1.0 结果: 3.0 False
预测: 1.0 结果: 3.0 False
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 3.0 False
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 2.0 False
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 3.0 False
预测: 1.0 结果: 1.0 True
预测: 1.0 结果: 2.0 False
Test Accuracy: 0.7205882352941176
Done ...

Process finished with exit code 0
```



0.70
0.65
0.60
0.55
0.50
0.45
0.40
accuracy
0
50
100
150
200
250
300
350
400
epochs
train
val



0.38
0.36
0.34
0.32
0.30
0.28
loss
0
50
100
150
200
250
300
350
400
epoch