BPNN 实验报告

516030910024 张政童

1. 实验内容:
   实现基本 后传递神经元网络 Back Propagation Neural Networks (BPNN) 算法

2. 实验环境
   a) python3.7
   b) VSCode
   c) Win10

3. 实现的功能:
   a) 1 输入层，N 隐藏层，1 输出层（N 可变）
   b) 隐藏层的节点数目可变
   c) 激活函数有三种
   d) 输入数据为矩阵
   e) 输入层维度与输出层维度均可变
   f) Learning Rate 与 limits 均可变

4. 用法:
   BBNP 类有五个常规参数:
   输入样例、期望样例、隐藏层节点数量、激励函数类型、隐藏层数量，
   以及两个缺省参数：迭代数量、learning rate，默认值分别为 10000 与 0.05
   在实例化之后，调用成员函数 test()即可进行迭代，并返回最终结果值，同时在 console 里打出 log

```python
def main():
    #init test data
    cases = [
            [0.5, 0.9, 0.1],
            [0.1, 0.7, 0.4],
            [0.99, 0.11, 0.3]
    ]
    expects =  [
        [0.1, 0.291, 0.7],
        [0.6, 0.8, 0.1],
        [0.1, 0.8, 0.81]
    ]
    number_of_neurons = 5
    function_mode = 2
    n_layers = 3
    limits = 100

    #init the main class with default learning-rate=0.05
    bn = BPNN(cases,expects, number_of_neurons, function_mode, n_layers, limits)
    bn.test()
```

5. 源码:

a) 主入口

```python
class BPNN:
    def __init__(self, cases, expects, nh, mode, nlayer, limits = 100000, learn = 0.05):
        if len(cases) < 1:
            print("case can not be empty!!")
            exit(0)
        if len(cases) != len(expects):
            print("cases and expects not matching !")
            exit(0)
        if limits <= 0 :
            print("limits should be a positive number!")
            exit(0)
        if learn<0 or learn >0.1:
            print("learning-rate should be between [0, 1]!")
            exit(0)
        self.input_n = len(cases[0]) + 1 #include bias
        self.hidden_n = nh + 1#include bias
        self.output_n = len(cases[0])
        self.mode = mode
        self.layer_n = nlayer
        self.cases = cases
        self.expects = expects
        self.limits = limits
        self.learn = learn
        #init datas
        self.input_datas = [1.0] * self.input_n
        self.hidden_datas = [1.0] * self.hidden_n
        self.output_datas = [1.0] * self.output_n
        self.ih_weights = []
        self.ho_weights = []
        #init weights
        for i in range(self.input_n):
            tmp = []
            for h in range(self.hidden_n - 1):
                tmp.append(rand(-0.2, 0.2))
                self.ih_weights.append(tmp)
        for h in range(self.hidden_n):
            tmp = []
            for o in range(self.output_n):
                tmp.append(rand(-0.2, 0.2))
                self.ho_weights.append(tmp)
```

自定义一个 BPNN 类，input、hidden、output 分别代表输入层、隐藏层与输出层三个 data 矩阵存放各个神经元节点的数据，初始化为 1.0；另外两个矩阵存放两个权重矩阵，初始化为(-0.2, 0.2)间的随机值

b) 激励函数选择：简单地使用 mode 来进行选择

```python
def sigmoid(x, mode):
    if mode == 1:
        return logistic(x)
    elif mode == 2:
        return tanh(x)
    elif mode == 3:
        return relu(x)
    else:
        print("mode does not exist!")
        return 0
def sigmoid_derivative(x, mode):
    if mode ==1:
        return logistic_derivative(x)
    elif mode ==2:
        return tanh_derivative(x)
    elif mode == 3:
        return relu_derivative(x)
    else:
        print("mode doex not exist")
```

c) Predict

进行加权计算，算出 hidden 层和 output 层每个节点的输出

```python
def predict(self, inputs):
    # activate input layer
    for i in range(self.input_n - 1):
        self.input_datas[i] = inputs[i]
    # activate hidden layer
    for j in range(self.hidden_n - 1):
        sum = 0.0
        for i in range(self.input_n):
            sum += self.input_datas[i] * self.ih_weights[i][j]
        self.hidden_datas[j] = sigmoid(sum, self.mode)
    # activate output layer
    for k in range(self.output_n):
        sum = 0.0
        for j in range(self.hidden_n - 1):
            sum += self.hidden_datas[j] * self.ho_weights[j][k]
        self.output_datas[k] = sigmoid(sum, self.mode)
    return self.output_datas
```

d) Back_propagate:

反向传播并更新权值

```python
def back_propagate(self, case, expect, learn ):
    #feed forward
    self.predict(case)
    #get output layer error
    output_errors = [0.0] * self.output_n
    for i in range(self.output_n):
        error = expect[i] - self.output_datas[i]
        output_errors[i] = sigmoid_derivative(self.output_datas[i], self.mode)*error
    #get hidden layer error
    hidden_errors = [0.0]*(self.hidden_n - 1)
    for i in range(self.hidden_n - 1):
        error = 0.0
        for j in range(self.output_n):
            error += output_errors[j] * self.ho_weights[i][j]
        hidden_errors[i] = sigmoid_derivative(self.hidden_datas[i], self.mode) * error
    #update ho_weights
    for i in range(self.hidden_n - 1):
        for j in range(self.output_n):
            change = output_errors[j] * self.hidden_datas[i]
            self.ho_weights[i][j] += learn * change
    #update ih_weights
    for i in range(self.input_n):
        for j in range(self.hidden_n - 1):
            change = hidden_errors[j] * self.input_datas[i]
            self.ih_weights[i][j] += learn * change
```

e) Train:

用于迭代测试

```python
def train(self, cases, expects):
    res = []
    for i in range(len(cases)):
        for j in range(self.limits):
            self.back_propagate(cases[i], expects[i], self.learn)
        #print("input:", self.input_datas)
        #print("expect:", expects[i])
        #print("result:", self.output_datas)
        tmp = copy.deepcopy(self.output_datas)
        res.append(tmp)
    return res
```

f) Test：

用于实现多个隐藏层的情况，每层处理完的输出结果当作下一层的输入

```
def test(self):
    cases = copy.deepcopy(self.cases)
    origin = copy.deepcopy(cases)
    expects = self.expects
    flag = self.layer_n
    while flag != 0 :
        flag -= 1
        res = self.train(cases, expects)
        #update the cases
        cases = copy.deepcopy(res)
        #print("new cases:", cases)
    print("cases:",origin)
    print("expect:", expects)
    print("result:", res)
    return res
```

6. 测试
   a) 默认迭代 10000 次，learning-rate 默认为 0.05
   b) 选取 3 个 layer，5 个 neuro，1000 次迭代，激励函数为 tanh 的情况：

```
cases: [[0.5, 0.9, 0.1], [0.1, 0.7, 0.4], [0.99, 0.11, 0.3]]
expect: [[0.1, 0.291, 0.7], [0.6, 0.8, 0.1], [0.1, 0.8, 0.81]]
result: [0.10003144929185592, 0.2911425893948132, 0.6997617719916662]
result: [0.6003567397367211, 0.7993341627730737, 0.10001873857122212]
result: [0.1000067221231542, 0.8009606390580444, 0.8090355099920381]
```

   根据以上数据可发现，期望数据与原始数据差别越小则迭代出的结果越精确，这一点与常识相符。同时另外两种激励函数得经过更多数量级的迭代才能达到如此精度，可能是算法有问题或者激励函数的缺陷，在此不做过多阐述。
7. 感悟

   这次用 python 简单地写了一个 BPNN 算法，通过网站查找资料，见过了许许多多不同的实现方法，虽然每个人的实现细节都不太一样，比如偏置节点的权重变或者不变、后传递时是否需要加入矫正率来使结果更精确等等，但是主要的思路都大同小异，以后可能会更多地用到该思想，希望通过这次实验能让我对机器学习有更深的理解。