《机器学习基础》实验报告

年级、专业、班级	2019 级计算机科学与技术卓越 02 班	姓名	李燕琴
实验题目	BP 算法实践		
实验时间	2021/11/07	实验地点	
实验成绩		实验性质	□验证性 □设计性 □综合性

教师评价：

□算法/实验过程正确； □源程序/实验内容提交 □程序结构/实验步骤合理；

□实验结果正确； □语法、语义正确； □报告规范；

其他：

评价教师签名：

一、实验目的

掌握 BP 算法原理并编程实践。

二、实验项目内容

1. 理解并**描述** BP 算法原理。

2. **编程**实践，将算法应用于合适的分类数据集（如鸢尾花、UCI 数据集、Kaggle 数据集），要求算法至少用于两个数据集。

三、实验过程或算法（源程序）

(1) BP 算法原理

基于《机器学习》-周志华，书中提到的三层神经网络模型，如图 1。我进行了多层神经网络模型的推导与计算，总览如图 2。其中Nerve[0]即输入的 X。

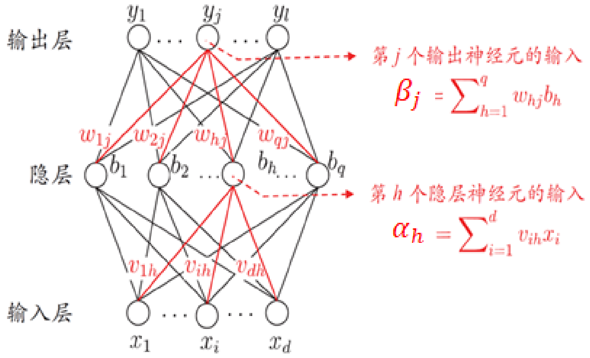


图 1 三层神经网络模型

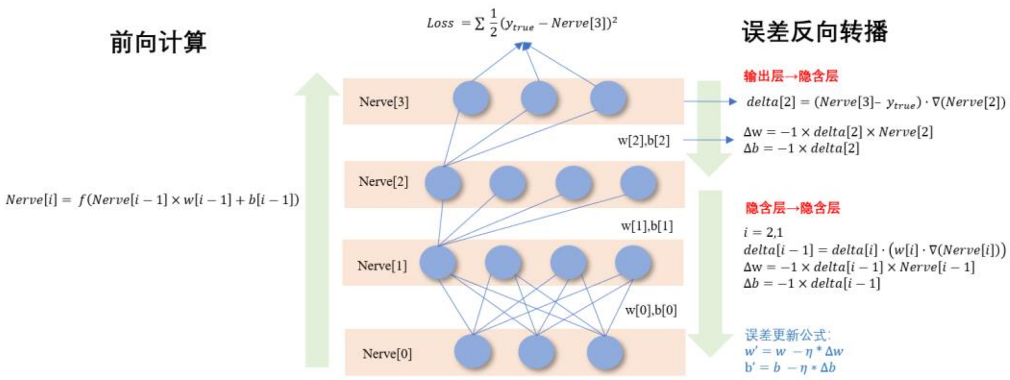


图 2 多层神经网络推导

①前向计算

1、神经网络的前一层和当前层，每一个神经元进行两两线性计算（即加权求和），再通过非线性激活函数f进行激活。

$$Nerve[i] = f(Nerve[i-1] \times w[i-1] + b[i-1])$$

2、输出层，为独热编码，通过激活函数，计算输出值和原始值的均方误差。

②反向传播

输出层→隐含层：

1、基于链式法则，利用前向传播最后输出的结果来计算其输出层损失函数的导数，即$\mathrm{y_{true}} - \mathrm{Nerve}[3]$，通过激活函数导数，计算其对输出层未激活值的偏导数。

$$delta[2] = (Nerve[3]- y_{true}) \cdot \nabla(Nerve[2])$$

2、利用残差结果，对前一层的隐藏层神经元值进行加权求和，得到该层参数w, b的增量

$$\Delta \mathrm{w} = -1 \times delta[2] \times Nerve[2]$$
$$\Delta b = -1 \times delta[2]$$

隐含层→隐含层/输出层：

1、残差计算，基于上一层神经元的残差，基于链式法则求导计算得到，

$$delta[i-1] = delta[i] \cdot \big(w[i] \cdot \nabla(Nerve[i])\big)$$

2、利用残差结果，对前一层的隐藏层神经元值进行加权求和，得到该层参数w, b的增量

$$\Delta \mathrm{w} = -1 \times delta[i-1] \times Nerve[i-1]$$
$$\Delta b = -1 \times delta[i-1]$$

如此一层一层的向后传下去直到输入层即可。最后根据误差更新公式逐层进行更新。

(2) 源代码

根据（1）BP 算法原理中的多层神经网络，误差反向传播原理，本实验基于 sklearn 机器学习逻辑和基本接口，实现了自定义 BpNet 类，如图 3 所示，其中为了使得 BpNet 更具有灵活性和用户友好性，本实验还实现了自定义隐含层网络层数，基于 batch 反向传播计算等。

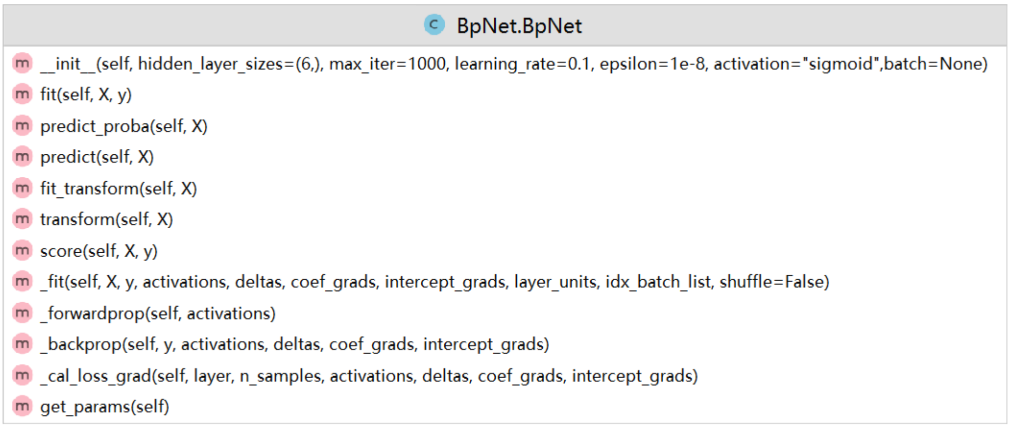


图 3 BpNet 方法接口

源代码如下：

```python
import time
import numpy as np
import pandas as pd
import random
import matplotlib
import matplotlib.pyplot as plt
from logging import error

matplotlib.rcParams['font.sans-serif'] = ['KaiTi']

def sigmoid(X):
    return 1 / (1 + np.exp(-X))


def sigmoid_diff(y):
    return y * (1 - y)


def tanh(X):
    return (np.exp(X) - np.exp(-X)) / (np.exp(X) + np.exp(-X))


def tanh_diff(y):
    return 1 - y ** 2
```

```python
def squared_loss(y_true, y_pred):
    return ((y_true - y_pred) ** 2).mean(axis=0).sum() / 2


def softmax(X):
    return np.exp(X) / np.sum(np.exp(X), axis=1).reshape(-1, 1)  # X / 按照行求和,得到(n_samples,1)矩
阵


def one_hot_encoder(y, class_encoder=None):
    if class_encoder == None:
        y_set = set(y.ravel())
        class_encoder = {label: idx for idx, label in enumerate(y_set)}
    n_classes = len(class_encoder)
    n_samples = len(y)
    y_one_hot = np.zeros((n_samples, n_classes), dtype=int) + 0.01
    for idx, label in enumerate(y.ravel()):
        y_one_hot[idx, class_encoder[label]] = 1 - 0.01
    return y_one_hot


def one_hot_decoder(y_one_hot, class_decoder=None):
    if class_decoder == None:
        class_decoder = {label: idx for idx, label in enumerate(range(y_one_hot.shape[1]))}
    y_transfer = y_one_hot.copy()
    for idx, col in enumerate(y_transfer.T):
        # 注意，这里的 col 只是 y_transfer 的一个视图
        col[col == 1] = class_decoder[idx]
    y = np.max(y_transfer, axis=1).astype(int)
    return y.reshape(-1, 1)  # [r,1]


ACTIVATIONS = {"sigmoid": sigmoid, "tanh": tanh}
DIFF = {"sigmoid": sigmoid_diff, "tanh": tanh_diff}


class BpNet:

    def __init__(self, hidden_layer_sizes=(6,), max_iter=1000, learning_rate=0.1, epsilon=1e-8,
activation="sigmoid",batch=None) -> None:
        '''
        BpNet 初始化
        @param hidden_layer_sizes:自定义隐含层元组
```

```
        @param max_iter:最大迭代次数
        @param learning_rate:学习率
        @param epsilon:最大误差精度
        @param activation:激活函数
        @param batch:batch size，例外情况：如果是负数，则为单样本训练结构；如果大于样本数，则直接全部训
练；
        '''
        self.hidden_layer_sizes = list(hidden_layer_sizes)
        self.max_iter = max_iter
        self.activation = activation
        self.learning_rate = learning_rate
        self.epsilon = epsilon
        self.batch = batch

    def fit(self, X, y):
        ''' 训练数据 '''
        # X 的预处理
        X = self.fit_transform(X)

        # Y 的维数判断
        if y.ndim == 1:
            y = y.reshape((-1, 1))
        # 转为独热编码
        if y.shape[1] == 1:
            y_set = set(y.ravel())
            # 经过测验，one_hot_encoder 和 one_hot_decoder 基本没有问题
            self.class_encoder = {label: idx for idx, label in enumerate(y_set)}
            self.class_decoder = {idx: label for idx, label in enumerate(y_set)}
            y = one_hot_encoder(y, self.class_encoder)

        # 层数设置
        n_samples, n_features = X.shape
        n_output = y.shape[1]
        layer_units = ([n_features] + self.hidden_layer_sizes + [n_output])
        self.layer_units = layer_units
        self.n_layers_ = len(layer_units)

        # 初始化 每一层的权重和阈值
        self.coef_ = []
        self.intercept_ = []
        for i in range(self.n_layers_ - 1):
            coef_init = np.random.random((layer_units[i], layer_units[i + 1]))
            intercept_init = np.random.random(layer_units[i + 1])
            self.coef_.append(coef_init)
```

```
            self.intercept_.append(intercept_init)

        # 每一层的 被激活的单元值
        activations = [X] + [None] * (len(layer_units) - 1)
        deltas = [None] * (len(activations) - 1)

        # 限定梯度计算
        coef_grads = [np.empty((n_in, n_out)) for n_in, n_out in zip(layer_units[:-1],
layer_units[1:])]
        intercept_grads = [np.empty(n_out) for n_out in layer_units[1:]]

        # 记录迭代次数
        self.n_iter_ = 0
        # 记录迭代样本数和损失函数
        self.loss_curve_ = []

        # 计算 batch
        if self.batch is None or self.batch > n_samples:
            self.batch = n_samples
        elif self.batch <= 0:
            self.batch = 1

        # 获取训练中需要的 idx_batch_list
        idx_batch_list = []
        last_idx = 0
        for idx in range(self.batch, n_samples + self.batch, self.batch):
            idx = min(idx, n_samples)
            idx_batch_list.append([last_idx, idx])
            last_idx = idx
        # print("训练结构: ",idx_batch_list)

        self._fit(X, y, activations, deltas, coef_grads, intercept_grads, layer_units,
idx_batch_list)

    def predict_proba(self, X):
        ''' 预测概率 '''
        if X.shape[1] != self.layer_units[0]:
            error("输入的 X", {X.shape}, "维数不正确")
            return False
        # X 要归一化
        X = self.transform(X)
        # 初始化神经网络层，确定 activations 每一个维度的大小
        activations = [X]
        for i in range(1, self.n_layers_):
```

```
            activations.append(np.empty((X.shape[0], self.layer_units[i])))
        # 前向传播，计算
        activations = self._forwardprop(activations)
        y_prob = activations[-1]
        return y_prob

    def predict(self, X):  # (n_samples,n_features)
        ''' 预测 '''
        y_prob = self.predict_proba(X)
        y_one_hot = np.zeros(y_prob.shape)
        y_max = np.argmax(y_prob, axis=1)
        for ridx, midx in enumerate(y_max):
            y_one_hot[ridx, midx] = 1
        return one_hot_decoder(y_one_hot, self.class_decoder)

    def fit_transform(self, X):
        ''' 记录 + X 的预处理：归一化 '''
        self.x_max = np.max(X, axis=0)
        self.x_min = np.min(X, axis=0)
        X = (X - self.x_min) / (self.x_max - self.x_min + 0.001)  # X / 按照行求和,得到(n_samples,1)
矩阵
        return X

    def transform(self, X):
        ''' X 的预处理：归一化 '''
        X = (X - self.x_min) / (self.x_max - self.x_min + 0.001)  # X / 按照行求和,得到(n_samples,1)
矩阵
        return X

    def score(self, X, y):
        ''' 预测准确率 '''
        # Y 的预处理
        if y.ndim == 1:
            y = y.reshape((-1, 1))
        y_pred = self.predict(X)
        return (y == y_pred).mean()

    def _fit(self, X, y, activations, deltas, coef_grads, intercept_grads, layer_units,
idx_batch_list, shuffle=False):
        n_samples = len(X)
        n_bp_cnt = len(idx_batch_list)
        sample_idx = np.arange(n_samples, dtype=int)
        # 开始迭代
        for it in range(self.max_iter):
```

```python
        accumulated_loss = 0.0
        if shuffle:
            random.shuffle(sample_idx)
        for idx_batch in idx_batch_list:
            # 获取训练样本
            li,ri = idx_batch
            sampleX, sampley = X[li:ri], y[li:ri]

            # 前向传播，计算预测值
            activations[0]= sampleX
            activations = self._forwardprop(activations)

            # 计算均方误差
            accumulated_loss += squared_loss(sampley, activations[-1])

            # 反向传播，计算梯度，更新权值和阈值
            self._backprop(sampley, activations, deltas, coef_grads, intercept_grads)

        self.n_iter_ += 1
        self.loss_curve_.append(accumulated_loss / n_bp_cnt)
        if self.loss_curve_[-1] < self.epsilon:
            break

def _forwardprop(self, activations):
    activation_fun = ACTIVATIONS[self.activation]

    # 逐层回归+激活
    for i in range(self.n_layers_ - 1):
        activations[i + 1] = np.dot(activations[i], self.coef_[i]) + self.intercept_[i]
        activations[i + 1] = activation_fun(activations[i + 1])

    # TODO 数据前推，尚未进行 softmax 处理
    # activations[i + 1] = softmax(activations[i + 1])

    return activations

def _backprop(self, y, activations, deltas, coef_grads, intercept_grads):
    diff_fun = DIFF[self.activation]
    n_samples = len(y)
    # 第一层没有残差计算，共 self.n_layers_-1 层计算残差，last = self.n_layers_ - 1 - 1，即最后一层
    last = self.n_layers_ - 2

    # TODO 反向传播，尚未进行 softmax 处理
    # 输出层->隐含层
```

```
        deltas[last] = (activations[-1] - y) * diff_fun(activations[-1])
        coef_grads, intercept_grads = self._cal_loss_grad(last, n_samples, activations, deltas,
coef_grads,
                                                          intercept_grads)
        # 隐含层->隐含层
        for i in range(last, 0, -1):
            deltas[i - 1] = np.dot(deltas[i], self.coef_[i].T) * diff_fun(activations[i])
            coef_grads, intercept_grads = self._cal_loss_grad(i - 1, n_samples, activations, deltas,
coef_grads,
                                                              intercept_grads)
        # TODO 学习器优化
        for i in range(self.n_layers_ - 1):
            self.coef_[i] += -1 * self.learning_rate * coef_grads[i]
            self.intercept_[i] += -1 * self.learning_rate * intercept_grads[i]
        return None

    def _cal_loss_grad(self, layer, n_samples, activations, deltas, coef_grads, intercept_grads):
        # deltas : 右层节点对应残差
        coef_grads[layer] = np.dot(activations[layer].T, deltas[layer])
        # coef_grads[layer] /= n_samples
        intercept_grads[layer] = np.mean(deltas[layer], axis=0)
        return coef_grads, intercept_grads

    def get_params(self):
        ''' 获取模型参数 '''
        return self.coef_, self.intercept_


def train_test_split(X, Y, train_percent=0.7, shuffle=True, seed=None):
    ''' 自定义数据分割 '''
    n_smaples = X.shape[0]
    if shuffle:
        idx = np.arange(n_smaples, dtype=int)
        if seed:
            random.seed(2)
        random.shuffle(idx)
        X = X[idx]
        Y = Y[idx]
    n_train = int(np.floor(n_smaples * train_percent))
    trainX, testX = X[0:n_train], X[n_train:-1]
    trainY, testY = Y[0:n_train], Y[n_train:-1]
    return trainX, testX, trainY, testY
```

```python
def get_iris_data(filepath='./iris.csv'):
    ''' 获取鸢尾花数据集 '''
    iris_df = pd.read_csv(filepath)
    iris_data = iris_df.values
    X = iris_data[:, :-1]
    Y = iris_data[:, -1][:, np.newaxis]
    return X, Y


def get_wine_data(filepath='./wine.data'):
    ''' 获取红酒数据集 '''
    wine_data = np.loadtxt(filepath, delimiter=",")
    Y = wine_data[:, 0][:, np.newaxis]
    X = wine_data[:, 1:]
    return X, Y


def get_digits_data():
    ''' 获取手写数字数据集 '''
    from sklearn.datasets import load_digits
    dig = load_digits()
    X = dig.data
    Y = dig.target
    return X, Y


def test(data_name):
    ''' 数据集统一测试 '''
    my_dataset = {
        "iris": {
            "name": "鸢尾花数据集",
            "get_fun": get_iris_data,
            "bp": BpNet(hidden_layer_sizes=(6, 4), max_iter=1000)
        },
        "wine": {
            "name": "红酒数据集",
            "get_fun": get_wine_data,
            "bp": BpNet(hidden_layer_sizes=(8, 6), max_iter=1000, batch=-1)
        }
    }

    print("=" * 30, my_dataset[data_name]["name"], "=" * 30)
    X, Y = my_dataset[data_name]["get_fun"]()
    trainX, testX, trainY, testY = train_test_split(X, Y, seed=1)
```

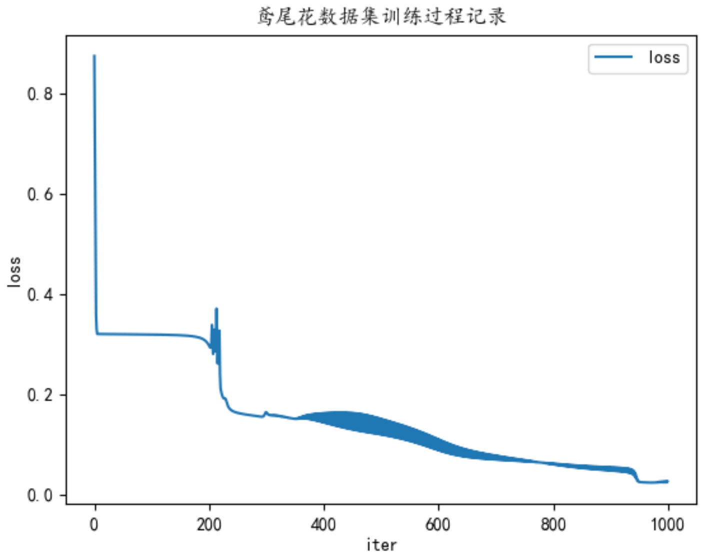
```python
    bp = my_dataset[data_name]["bp"]  # type: BpNet

    start_time = time.time()
    print("模型开始训练")
    bp.fit(trainX, trainY)
    print("模型结构：", bp.layer_units)
    print("模型训练结束，用时%.3fs" % ((time.time() - start_time) / 60))

    # 绘制训练过程
    plt.plot(range(len(bp.loss_curve_)), bp.loss_curve_)
    plt.title(my_dataset[data_name]["name"] + "训练过程记录")
    plt.legend(['loss'])
    plt.xlabel("iter")
    plt.ylabel("loss")
    plt.show()

    # 预测和评估
    # print("真实值：",testY.ravel())
    # print("预测值：",bp.predict(testX).ravel())
    print("测试集：")
    predY = bp.predict_proba(testX)
    print("损失函数值：%.3f" % (squared_loss(testY, predY)))
    print("预测准确率：%.3f" % (bp.score(testX, testY)))


if __name__ == "__main__":
    test("iris")
    test("wine")
```

四、实验结果及分析

根据实验要求，本实验测试了 iris 数据集（源自 sklearn 数据集）和 wine 数据集（源自 uci 数据集）。根据 train:test=7:3，划分训练集和测试集，其中结果如下，可以看到鸢尾花的准确率能达到 100%，红酒数据集的准确率能达到 94%，说明自定义的 BpNet 具有逻辑无误，且反向传播原理具有较好的分类效果：

(1) 鸢尾花数据集

```
============================== 鸢尾花数据集 ==============================
模型开始训练
模型结构： [4, 6, 4, 3]
模型训练结束，用时0.006s
测试集：
损失函数值：0.000
预测准确率：1.000
```



(2) 红酒数据集

============================== 红酒数据集 ==============================

模型开始训练

模型结构： [13, 8, 6, 3]

模型训练结束，用时0.399s

测试集：

损失函数值：0.009

预测准确率：0.981