《机器学习基础》实验报告

年级、专业、班级	2021 级计算机科学与技术 4 班	姓名	胡 鑫
实验题目	BP 算法实践		
实验时间	2023 年 4 月 27 日	实验地点	DS3402
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。

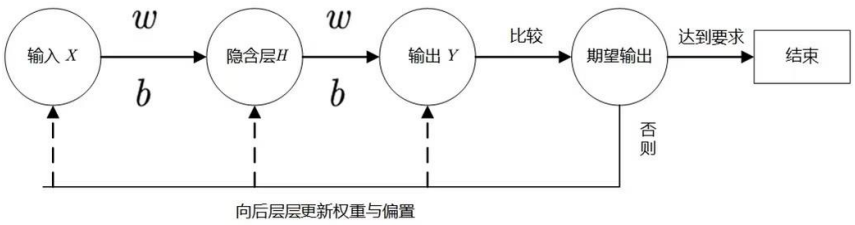
二、实验项目内容

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。

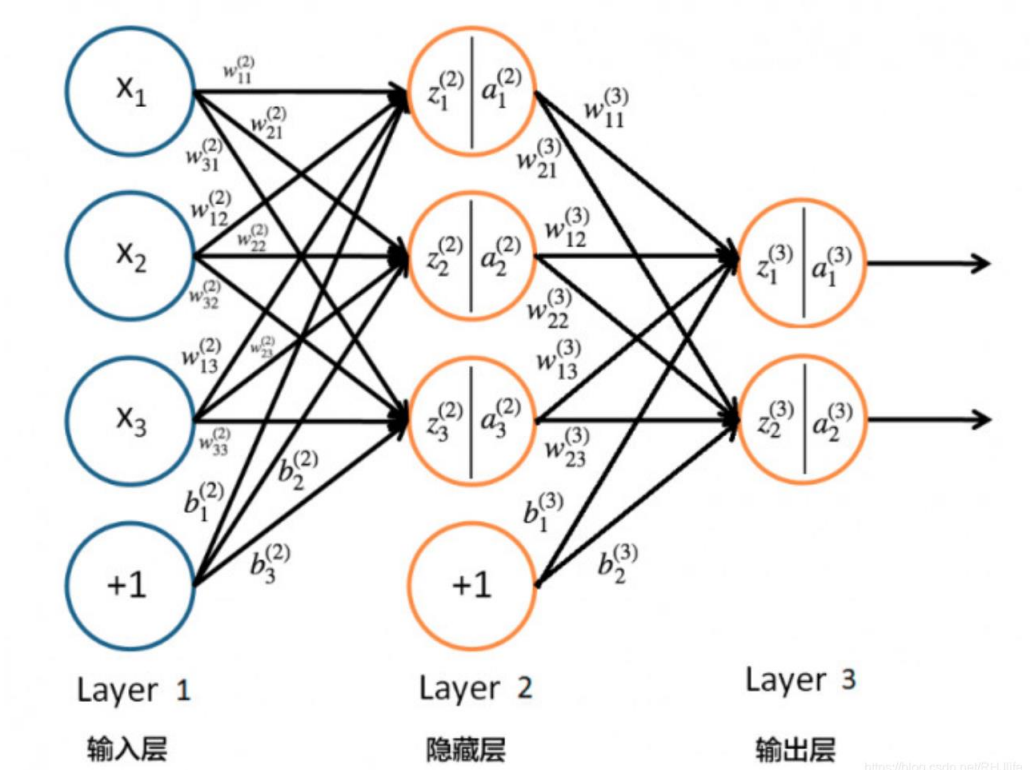
三、实验过程或算法（源程序）

1.BP 算法原理的全称是误差反向传播算法，它的基本思想是，先计算每一层的状态和激活值，直到最后一层，此时信号是前向传播的；然后计算每一层的误差，误差的计算过程是从最后一层向前推进，这也是反向传播算法的由来；最后是更新参数，使得网络的实际输出值和期望输出值的误差均方差最小。然后迭代前两个步骤，直到满足停止准则，比如一定的迭代次数或者是一定的损失。BP 神经网络的主要特点是：信号是前向传播的，而误差是反向传播的。



以三层的感知器为例，第一层是隐藏层，第二层是隐藏层，第三层是输出层，隐藏层的输入由输入层和权重矩阵 w 和偏置向量 b 共同决定，隐藏层

的输出由神经元的激活函数和输入决定；最后输出层的输入和输出，也和隐藏层类似，输出层常用 sigmoid 函数作为激活函数。在误差反向传播的过程中，代价函数仅和权重矩阵和偏置向量有关，调整权重和偏置可以减少或增大代价。



BP 算法核心公式就是求某个训练数据的代价函数对参数的偏导数。具体应用是，第一步，初始化参数 W，b 为一个很小的，接近于零的随机值，第二步，利用前向传播公式 z=Wa+b 和 a=f（z）计算每层的状态和激活值，第三步是计算计算输出层的误差，然后再从后往前计算隐藏层的误差，然后就可以求这个训练数据的代价函数对参数的偏导数，然后再根据参数变化率对参数进行更新。

2.（1）鸢尾花数据集

```
import pandas as pd
import numpy as np

# 1.初始化参数
def init_parameters(n_x, n_h, n_y):
    np.random.seed(2)

    #  权重和偏置矩阵
    w1 = np.random.randn(n_h, n_x) * 0.01
    #使用 NumPy 库生成形状为(n_h, n_x)的随机矩阵，每个元素都是从标准正态分布中随机抽取的。
    #  乘以 0.01 是为了将矩阵中的元素缩小，以便更好地进行后续计算。
    b1 = np.zeros(shape=(n_h, 1))
    w2 = np.random.randn(n_y, n_h) * 0.01
```

```
    b2 = np.zeros(shape=(n_y, 1))

    # 通过字典存储参数
    parameters = {'w1': w1, 'b1': b1, 'w2': w2, 'b2': b2}

    return parameters


# 2.前向传播
def forward_propagation(X, parameters):
    w1 = parameters['w1']
    b1 = parameters['b1']
    w2 = parameters['w2']
    b2 = parameters['b2']

    # 通过前向传播来计算 a2
    z1 = np.dot(w1, X) + b1
    #这段代码是使用 NumPy 库中的 dot 函数计算矩阵 w1 和矩阵 X 的乘积，然后加上偏置向量 b1。
    # 结果是一个形状为(n_h, m)的矩阵，n_h 是隐藏层的神经元数量，m 是输入样本的数量。
    # 这个地方需注意矩阵加法：虽然(w1*X)和 b1 的维度不同，但可以相加
    a1 = np.tanh(z1)                    # 使用 tanh 作为第一层的激活函数
    z2 = np.dot(w2, a1) + b2
    a2 = 1 / (1 + np.exp(-z2))    # 使用 sigmoid 作为第二层的激活函数

    # 通过字典存储参数
    cache = {'z1': z1, 'a1': a1, 'z2': z2, 'a2': a2}

    return a2, cache


# 3.计算代价函数
def compute_cost(a2, Y, parameters):
    m = Y.shape[1]          # Y 的列数即为总的样本数

    # 采用交叉熵（cross-entropy）作为代价函数
    logprobs = np.multiply(np.log(a2), Y) + np.multiply((1 - Y), np.log(1 - a2))
    cost = - np.sum(logprobs) / m

    return cost
```

```
# 4.反向传播（计算代价函数的导数）
def back_propagation(parameters, cache, X, Y):
    m = Y.shape[1]

    w2 = parameters['w2']

    a1 = cache['a1']
    a2 = cache['a2']

    # 反向传播，计算 dw1、db1、dw2、db2
    dz2 = a2 - Y
    dw2 = (1 / m) * np.dot(dz2, a1.T)
    db2 = (1 / m) * np.sum(dz2, axis=1, keepdims=True)

    dz1 = np.multiply(np.dot(w2.T, dz2), 1 - np.power(a1, 2))
    dw1 = (1 / m) * np.dot(dz1, X.T)
    db1 = (1 / m) * np.sum(dz1, axis=1, keepdims=True)

    grads = {'dw1': dw1, 'db1': db1, 'dw2': dw2, 'db2': db2}

    return grads


# 5.更新参数
def update_parameters(parameters, grads, learning_rate=0.4):
    w1 = parameters['w1']
    b1 = parameters['b1']
    w2 = parameters['w2']
    b2 = parameters['b2']

    dw1 = grads['dw1']
    db1 = grads['db1']
    dw2 = grads['dw2']
    db2 = grads['db2']

    # 更新参数
    w1 = w1 - dw1 * learning_rate
    b1 = b1 - db1 * learning_rate
    w2 = w2 - dw2 * learning_rate
    b2 = b2 - db2 * learning_rate

    parameters = {'w1': w1, 'b1': b1, 'w2': w2, 'b2': b2}

    return parameters
```

```
# 6.模型评估
def predict(parameters, x_test, y_test):
    w1 = parameters['w1']
    b1 = parameters['b1']
    w2 = parameters['w2']
    b2 = parameters['b2']

    z1 = np.dot(w1, x_test) + b1
    a1 = np.tanh(z1)
    z2 = np.dot(w2, a1) + b2
    a2 = 1 / (1 + np.exp(-z2))

    # 结果的维度
    n_rows = y_test.shape[0]
    n_cols = y_test.shape[1]

    # 预测值结果存储
    output = np.empty(shape=(n_rows, n_cols), dtype=int)

    for i in range(n_rows):
        for j in range(n_cols):
            if a2[i][j] > 0.5:
                output[i][j] = 1
            else:
                output[i][j] = 0

    print('预测结果：')
    print(output)
    print('真实结果：')
    print(y_test)

    count = 0
    for k in range(0, n_cols):
        if output[0][k] == y_test[0][k] and output[1][k] == y_test[1][k] and
output[2][k] == y_test[2][k]:
            count = count + 1
        else:
            print('第 %d 列与真实结果不同' % k)

    acc = count / int(y_test.shape[1]) * 100
    print('准确率：%.2f%%' % acc)
    return output
```

```
# 建立神经网络
def nn_model(X, Y, n_h, n_input, n_output, num_iterations=13000, print_cost=False):
    np.random.seed(3)

    n_x = n_input                # 输入层节点数
    n_y = n_output               # 输出层节点数

    # 1.初始化参数
    parameters = init_parameters(n_x, n_h, n_y)

    # 梯度下降循环
    for i in range(0, num_iterations):
        # 2.前向传播
        a2, cache = forward_propagation(X, parameters)
        # 3.计算代价函数
        cost = compute_cost(a2, Y, parameters)
        # 4.反向传播
        grads = back_propagation(parameters, cache, X, Y)
        # 5.更新参数
        parameters = update_parameters(parameters, grads)

        # 每 1000 次迭代，输出一次代价函数
        if print_cost and i % 1000 == 0:
            print('迭代第%i 次，代价函数为：%f' % (i, cost))

    return parameters


if __name__ == "__main__":
    # 读取数据
    data_set = pd.read_csv('iris_training.csv', header=None)#相对路径

    X = data_set.iloc[:, 0:4].values.T          # 前四列是特征，T 表示转置
    Y = data_set.iloc[:, 4:].values.T           # 后三列是标签
    Y = Y.astype('uint8')

    # 开始训练
    # 输入 4 个节点，隐层 20 个节点，输出 3 个节点，迭代 13000 次
    parameters = nn_model(X, Y, n_h=20, n_input=4, n_output=3, num_iterations=13000, print_cost=True)
```

```
    # 对模型进行测试
    data_test = pd.read_csv('iris_test.csv', header=None)
    x_test = data_test.iloc[:, 0:4].values.T
    y_test = data_test.iloc[:, 4:].values.T
    y_test = y_test.astype('uint8')

    result = predict(parameters, x_test, y_test)
```

（2）

```
import pandas as pd
import numpy as np

# 1.初始化参数
def init_parameters(n_x, n_h1,n_h2, n_y):
    np.random.seed(3)

    # 权重和偏置矩阵
    w1 = np.random.randn(n_h1, n_x) * 0.01
    b1 = np.zeros(shape=(n_h1, 1))
    w2 = np.random.randn(n_h2, n_h1) * 0.01
    b2 = np.zeros(shape=(n_h2, 1))
    w3 = np.random.randn(n_y, n_h2) * 0.01
    b3 = np.zeros(shape=(n_y, 1))
    # 用字典存储参数
    parameters = {'w1': w1, 'b1': b1, 'w2': w2, 'b2': b2,'w3': w3, 'b3': b3}

    return parameters


# 2.前向传播
def forward_propagation(X, parameters):
    w1 = parameters['w1']
    b1 = parameters['b1']
    w2 = parameters['w2']
    b2 = parameters['b2']
    w3 = parameters['w3']
    b3 = parameters['b3']

    # 通过前向传播来计算 a2
    z1 = np.dot(w1, X) + b1
    a1 = np.tanh(z1)                # 使用 tanh 作为第一层的激活函数
    z2 = np.dot(w2, a1) + b2
    a2 = np.tanh(z2)                # 使用 tanh 作为第二层的激活函数
    z3 = np.dot(w3, a2) + b3
```

```
    a3 = 1 / (1 + np.exp(-z3))    # 使用 sigmoid 作为第三层的激活函数

    # 通过字典存储参数
    cache = {'z1': z1, 'a1': a1, 'z2': z2, 'a2': a2,'z3': z3, 'a3': a3}

    return a3, cache


# 3.计算代价函数
def compute_cost(a3, Y, parameters):
    m = Y.shape[1]          # Y 的列数即为总的样本数

    # 采用交叉熵（cross-entropy）作为代价函数
    logprobs = np.multiply(np.log(a3), Y) + np.multiply((1 - Y), np.log(1 - a3))
    cost = - np.sum(logprobs) / m

    return cost


# 4.反向传播
def back_propagation(parameters, cache, X, Y):
    m = Y.shape[1]

    w2 = parameters['w2']
    w3 = parameters['w3']
    a1 = cache['a1']
    a2 = cache['a2']
    a3 = cache['a3']

    dz3 = a3 - Y
    dw3 = (1 / m) * np.dot(dz3, a2.T)
    db3 = (1 / m) * np.sum(dz3, axis=1, keepdims=True)

    dz2 = np.multiply(np.dot(w3.T, dz3), 1 - np.power(a2, 2))
    dw2 = (1 / m) * np.dot(dz2, a1.T)
    db2 = (1 / m) * np.sum(dz2, axis=1, keepdims=True)

    dz1 = np.multiply(np.dot(w2.T, dz2), 1 - np.power(a1, 2))
    dw1 = (1 / m) * np.dot(dz1, X.T)
    db1 = (1 / m) * np.sum(dz1, axis=1, keepdims=True)

    grads = {'dw1': dw1, 'db1': db1, 'dw2': dw2, 'db2': db2,'dw3': dw3, 'db3':
db3}
```

```
    return grads


# 5.更新参数
def update_parameters(parameters, grads, learning_rate=0.4):
    w1 = parameters['w1']
    b1 = parameters['b1']
    w2 = parameters['w2']
    b2 = parameters['b2']
    w3 = parameters['w3']
    b3 = parameters['b3']

    dw1 = grads['dw1']
    db1 = grads['db1']
    dw2 = grads['dw2']
    db2 = grads['db2']
    dw3 = grads['dw3']
    db3 = grads['db3']

    #  更新参数
    w1 = w1 - dw1 * learning_rate
    b1 = b1 - db1 * learning_rate
    w2 = w2 - dw2 * learning_rate
    b2 = b2 - db2 * learning_rate
    w3 = w3 - dw3 * learning_rate
    b3 = b3 - db3 * learning_rate
    parameters = {'w1': w1, 'b1': b1, 'w2': w2, 'b2': b2,'w3': w3, 'b3': b3}

    return parameters


# 6.模型评估
def predict(parameters, x_test, y_test):
    w1 = parameters['w1']
    b1 = parameters['b1']
    w2 = parameters['w2']
    b2 = parameters['b2']
    w3 = parameters['w3']
    b3 = parameters['b3']

    z1 = np.dot(w1, x_test) + b1
    a1 = np.tanh(z1)
    z2 = np.dot(w2, a1) + b2
    a2 = np.tanh(z2)
```

```
    z3 = np.dot(w3, a2) + b3
    a3 = 1 / (1 + np.exp(-z3))

    # 结果的维度
    n_rows = y_test.shape[0]
    n_cols = y_test.shape[1]

    # 预测值结果存储
    output = np.empty(shape=(n_rows, n_cols), dtype=int)

    for i in range(n_rows):
        for j in range(n_cols):
            if a3[i][j] > 0.5:
                output[i][j] = 1
            else:
                output[i][j] = 0

    print('预测结果：')
    print(output)
    print('真实结果：')
    print(y_test)

    count = 0
    for k in range(0, n_cols):
        if output[0][k] == y_test[0][k] and output[1][k] == y_test[1][k]:
            count = count + 1
        else:
            print('第 %d 列与真实结果不同' % k)

    acc = count / int(y_test.shape[1]) * 100
    print('准确率：%.2f%%' % acc)
    return output


# 建立神经网络
def nn_model(X, Y, n_h1, n_h2, n_input, n_output, num_iterations=1000,
print_cost=False):
    np.random.seed(3)

    n_x = n_input                    # 输入层节点数
    n_y = n_output                   # 输出层节点数

    # 1.初始化参数
    parameters = init_parameters(n_x, n_h1, n_h2, n_y)
```

```
    # 梯度下降循环
    for i in range(0, num_iterations):
        # 2.前向传播
        a3, cache = forward_propagation(X, parameters)
        # 3.计算代价函数
        cost = compute_cost(a3, Y, parameters)
        # 4.反向传播
        grads = back_propagation(parameters, cache, X, Y)
        # 5.更新参数
        parameters = update_parameters(parameters, grads)

        # 每 1000 次迭代，输出一次代价函数
        if print_cost and i % 100 == 0:
            print('迭代第%i 次，代价函数为：%f' % (i, cost))

    return parameters


if __name__ == "__main__":
    # 读取数据
    data_set = pd.read_csv('heart_training.csv', header=None)#相对路径

    X = data_set.iloc[:, 0:13].values.T          # 前十三列是特征，T 表示转置
    Y = data_set.iloc[:, 13:].values.T           # 后两列是标签
    Y = Y.astype('uint8')

    # 开始训练
    parameters = nn_model(X, Y, n_h1=10, n_h2=5, n_input=13, n_output=2, num_iterations=1000, print_cost=True)

    # 模型测试
    data_test = pd.read_csv('heart_test.csv', header=None)
    x_test = data_test.iloc[:, 0:13].values.T
    y_test = data_test.iloc[:, 13:].values.T
    y_test = y_test.astype('uint8')

    result = predict(parameters, x_test, y_test)
```

（3）红酒质量

```
import pandas as pd
import numpy as np

# 1.初始化参数
```

```
def init_parameters(n_x, n_h, n_y):
    np.random.seed(2)

    # 权重和偏置矩阵
    w1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros(shape=(n_h, 1))
    w2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros(shape=(n_y, 1))

    # 通过字典存储参数
    parameters = {'w1': w1, 'b1': b1, 'w2': w2, 'b2': b2}

    return parameters


# 2.前向传播
def forward_propagation(X, parameters):
    w1 = parameters['w1']
    b1 = parameters['b1']
    w2 = parameters['w2']
    b2 = parameters['b2']

    # 通过前向传播来计算 a2
    z1 = np.dot(w1, X) + b1
    a1 = np.tanh(z1)                    # 使用 tanh 作为第一层的激活函数
    z2 = np.dot(w2, a1) + b2
    a2 = 1 / (1 + np.exp(-z2))    # 使用 sigmoid 作为第二层的激活函数

    # 通过字典存储参数
    cache = {'z1': z1, 'a1': a1, 'z2': z2, 'a2': a2}

    return a2, cache


# 3.计算代价函数
def compute_cost(a2, Y, parameters):
    m = Y.shape[1]          # Y 的列数即为总的样本数

    # 采用交叉熵（cross-entropy）作为代价函数
    logprobs = np.multiply(np.log(a2), Y) + np.multiply((1 - Y), np.log(1 - a2))
    cost = - np.sum(logprobs) / m

    return cost
```

```
# 4.反向传播（计算代价函数的导数）
def back_propagation(parameters, cache, X, Y):
    m = Y.shape[1]

    w2 = parameters['w2']

    a1 = cache['a1']
    a2 = cache['a2']

    # 反向传播，计算 dw1、db1、dw2、db2
    dz2 = a2 - Y
    dw2 = (1 / m) * np.dot(dz2, a1.T)
    db2 = (1 / m) * np.sum(dz2, axis=1, keepdims=True)

    dz1 = np.multiply(np.dot(w2.T, dz2), 1 - np.power(a1, 2))
    dw1 = (1 / m) * np.dot(dz1, X.T)
    db1 = (1 / m) * np.sum(dz1, axis=1, keepdims=True)

    grads = {'dw1': dw1, 'db1': db1, 'dw2': dw2, 'db2': db2}

    return grads


# 5.更新参数
def update_parameters(parameters, grads, learning_rate=0.4):
    w1 = parameters['w1']
    b1 = parameters['b1']
    w2 = parameters['w2']
    b2 = parameters['b2']

    dw1 = grads['dw1']
    db1 = grads['db1']
    dw2 = grads['dw2']
    db2 = grads['db2']

    # 更新参数
    w1 = w1 - dw1 * learning_rate
    b1 = b1 - db1 * learning_rate
    w2 = w2 - dw2 * learning_rate
    b2 = b2 - db2 * learning_rate

    parameters = {'w1': w1, 'b1': b1, 'w2': w2, 'b2': b2}
```

```
    return parameters


# 6.模型评估
def predict(parameters, x_test, y_test):
    w1 = parameters['w1']
    b1 = parameters['b1']
    w2 = parameters['w2']
    b2 = parameters['b2']

    z1 = np.dot(w1, x_test) + b1
    a1 = np.tanh(z1)
    z2 = np.dot(w2, a1) + b2
    a2 = 1 / (1 + np.exp(-z2))

    #  结果的维度
    n_rows = y_test.shape[0]
    n_cols = y_test.shape[1]

    #  预测值结果存储
    output = np.empty(shape=(n_rows, n_cols), dtype=int)

    for i in range(n_rows):
        for j in range(n_cols):
            if a2[i][j] > 0.5:
                output[i][j] = 1
            else:
                output[i][j] = 0

    print('预测结果：')
    print(output)
    print('真实结果：')
    print(y_test)

    count = 0
    for k in range(0, n_cols):
        if output[0][k] == y_test[0][k] and output[1][k] == y_test[1][k] and
output[2][k] == y_test[2][k] and output[0][k] == y_test[0][k] and output[3][k]
== y_test[3][k] and output[4][k] == y_test[4][k] and output[5][k] ==
y_test[5][k]:
            count = count + 1
        else:
            print('第 %d 列与真实结果不同' % k)
```

```
    acc = count / int(y_test.shape[1]) * 100
    print('准确率：%.2f%%' % acc)
    return output


# 建立神经网络
def nn_model(X, Y, n_h, n_input, n_output, num_iterations=7700, print_cost=False):
    np.random.seed(3)

    n_x = n_input                    # 输入层节点数
    n_y = n_output                   # 输出层节点数

    # 1.初始化参数
    parameters = init_parameters(n_x, n_h, n_y)

    # 梯度下降循环
    for i in range(0, num_iterations):
        # 2.前向传播
        a2, cache = forward_propagation(X, parameters)
        # 3.计算代价函数
        cost = compute_cost(a2, Y, parameters)
        # 4.反向传播
        grads = back_propagation(parameters, cache, X, Y)
        # 5.更新参数
        parameters = update_parameters(parameters, grads)

        # 每 1000 次迭代，输出一次代价函数
        if print_cost and i %500 == 0:
            print('迭代第%i 次，代价函数为：%f' % (i, cost))

    return parameters


if __name__ == "__main__":
    # 读取数据
    data_set = pd.read_csv('redwinequality_training.csv', header=None)#相对路径

    X = data_set.iloc[:, 0:11].values.T              # 前十一列是特征，T 表示转置
    Y = data_set.iloc[:, 11:].values.T               # 后六列是标签
    Y = Y.astype('uint8')
```

```
    parameters = nn_model(X, Y, n_h=23, n_input=11, n_output=6, num_iterations=7800, print_cost=True)

    # 对模型进行测试
    data_test = pd.read_csv('redwinequality_test.csv', header=None)
    x_test = data_test.iloc[:, 0:11].values.T
    y_test = data_test.iloc[:, 11:].values.T
    y_test = y_test.astype('uint8')

    result = predict(parameters, x_test, y_test)
```

四、实验结果及分析

1．在鸢尾花数据集中，代价函数波动下降，在经过 12000 次迭代后，代价函数为 0.080251，测试的准确率为 96.67%

```
PS D:\ALL_Project\VSCode_Files\PythonCode\Lab2> & D:/Python-3.11.2/python.exe d:/ALL_Project/VSCode_Files/PythonCode/Lab2/Iris_BP.py
迭代第0次，代价函数为：2.080125
迭代第1000次，代价函数为：1.453051
迭代第2000次，代价函数为：0.222059
迭代第3000次，代价函数为：0.134724
迭代第4000次，代价函数为：0.118522
迭代第5000次，代价函数为：0.108532
迭代第6000次，代价函数为：0.120532
迭代第7000次，代价函数为：0.109784
迭代第8000次，代价函数为：0.097330
迭代第9000次，代价函数为：0.090706
迭代第10000次，代价函数为：0.086862
迭代第11000次，代价函数为：0.084378
迭代第12000次，代价函数为：0.080251
预测结果：
[[0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 1 0 1 1 0 1 0 0 0 0 0 1 0 0 0]
 [1 0 0 1 1 1 0 0 1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 0 1 1 0 1 0 1]
 [0 1 0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 1 0]]
真实结果：
[[0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 1 0 1 1 0 1 0 0 0 0 0 1 0 0 0]
 [1 0 0 1 1 1 0 0 1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 1 1 1 0 1 0 1]
 [0 1 0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0]]
第 23 列与真实结果不同
准确率：96.67%
```

在心脏病数据集中，代价函数下降缓慢，在经过 1000 次迭代后最终维持在 1.37164 附近，准确率为 81.08%，与鸢尾花数据集比较，有明显的下降。

```
PS D:\ALL_Project\VSCode_Files\PythonCode\Lab2> & D:/Python-3.11.2/python.exe d:/ALL_Project/V
SCode_Files/PythonCode/Lab2/heart_BP.py
迭代第0次，代价函数为：1.386313
迭代第100次，代价函数为：1.371646
迭代第200次，代价函数为：1.371647
迭代第300次，代价函数为：1.371652
迭代第400次，代价函数为：1.371646
迭代第500次，代价函数为：1.371646
迭代第600次，代价函数为：1.371646
迭代第700次，代价函数为：1.371641
迭代第800次，代价函数为：1.371647
迭代第900次，代价函数为：1.371648
预测结果：
[[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0]]
真实结果：
[[1 1 1 1 0 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1
  1]
 [0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0
  0]]
第 4 列与真实结果不同
第 7 列与真实结果不同
第 12 列与真实结果不同
第 17 列与真实结果不同
第 23 列与真实结果不同
第 28 列与真实结果不同
第 33 列与真实结果不同
准确率：81.08%
```

在选取的部分红酒质量数据集上使用该算法，迭代 7400 次后，代价函数为维持在 1.57 附近，准确率为 74.19%，相较之前两个数据集有所下降。

```
调试控制台  输出  问题  终端                                    Python

PS D:\ALL_Project\VSCode_Files\PythonCode\Lab2> & D:/Python-3.11.2/python.exe d:/ALL_Project/V
迭代第2000次，代价函数为：1.762761
迭代第2500次，代价函数为：1.716401
迭代第3000次，代价函数为：1.742162
迭代第3500次，代价函数为：1.749660
迭代第4000次，代价函数为：1.795693
迭代第4500次，代价函数为：1.659804
迭代第5000次，代价函数为：1.621315
迭代第5500次，代价函数为：1.616633
迭代第6000次，代价函数为：1.616736
迭代第6500次，代价函数为：1.591513
迭代第7000次，代价函数为：1.570035
预测结果：
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0]
 [0 1 1 1 0 0 1 0 1 1 1 0 0 1 0 0 1 1 0 1 1 0 1 1 1 0 1 1 1 0 0 1 1 1 0 0 1 1 1 0 1 1 1 0 0 0
  0]
 [0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
  0]
 [1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
  1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0]]
真实结果：
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0]
 [0 1 1 1 0 0 1 0 1 1 0 0 0 1 1 0 0 1 0 0 1 1 1 0 1 0 1 0 1 0 0 1 1 1 0 0 1 1 1 0 1 1 1 0 0 0
  0]
 [0 0 0 0 1 0 0 1 0 0 1 1 1 0 0 1 1 0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 1 1 0 0 1 1 1 0 1 1 1 0 0 0
  0]
 [1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
  1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0]]
第 4 列与真实结果不同
第 10 列与真实结果不同
第 14 列与真实结果不同
第 16 列与真实结果不同
第 19 列与真实结果不同
第 21 列与真实结果不同
第 23 列与真实结果不同
第 27 列与真实结果不同
准确率：74.19%
```

2.遇到的问题及解决办法：

（1）首先是在将该算法运用到心脏病数据集上的时候，准确率为 32%，训练的效果并不好。在添加了一层以 tanh 为激活函数的隐藏层后，准确率变化不明显，于是我多次更改迭代次数和隐藏层神经元个数，最终的准确

率有了显著提升，代价函数也明显下降。
（2）在运用到红酒质量数据集上的时候，也有和心脏病数据集相同的问题。但是在更改迭代次数和隐藏层神经元个数后，就有了明显改善。在此基础上，我结合 Excel 软件，随机选取了部分红酒质量数据集分别作为训练数据和测试数据，以排除人为因素对数据的干扰以及数据本身的影响，由此，准确率有了很大提升。