**《机器学习基础》实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | |  | | | **姓名** |  |
| **实验题目** | **BP算法实践** | | | | | |
| **实验时间** | **2020/12/08** | | **实验地点** | **DS3 303** | | |
| **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.理解并描述BP算法原理。**  （1）神经网络基本原理：  神经网络主要由输入层、隐含层和输出层组成。当只有一个隐含层时，该网络是一个两层的神经网络。由于输入层没有进行任何转换，所以不能将其视为一个单一的功能层。在实践中，网络输入层中的每个神经元代表一个属性特征。  使用神经网络进行分类任务时，输出层的数量代表分类标签的数量。而隐藏层神经元数和隐藏层神经元数是手动设置的。本实验中选取合适的隐藏层数与每层的神经元数以达到良好的分类效果。  （2）BP算法基本原理：  与普通分类器不同的是，神经网络是一个巨大的网络，最后一层的输出与每一层的神经元有关。神经网络的每一层与下一层之间，都有一个参数矩阵，称为权重矩阵，用来描述每个节点之间的值的关系。我们需要通过优化算法求解每一层的参数矩阵。对于有K层的神经网络，我们需要求解K−1参数矩阵。因此，我们不能直接计算目标函数的梯度来求解参数矩阵。使用BP算法对于神经网络进行优化分为两步，分别为“正向传播--fp”与“逆向传播—bp”。  （3）正向传播：  正向传播是计算从输入层到输出层每一层神经元的激活值。也就是说，先随机初始化每一层的参数矩阵，然后从输入层开始依次计算下一层每一个神经元的输出值（激活值），直到最后计算出输出层神经元的输出值（激活值）。正向传播算法的步骤如下：   1. 初始化权重矩阵W与偏置   初始化权重（weights）和偏置（bias）：随机初始化在-1到1之间，或者-0.5到0.5之间，每个单元有一个偏置。   1. 计算每个隐藏层的每个神经元输出值 2. 计算输出层的每个神经元输出值   （4）逆向传播：  反向传播是根据前向传播计算出来的激活值，来计算每一层参数的梯度，并根据梯度，使用最速下降法从后往前进行参数的更新。   1. 损失函数   损失函数定义为     1. 正则化：在误差目标函数中增加一项描述网络复杂程度的部分, 例如连接权值与阈值的平方和   其中损失函数加法的第二项是连接权值与阈值的平方和，使用这一项进行正则化，缓解神经网络过拟合对结果产生的影响。   1. 计算每一层的损失 2. 计算每一层的导函数 3. 计算每一个参数的梯度 4. 利用实验一中使用到的梯度最速下降法，对参数进行跟更新。（算法与实验一中一致，此处出于篇幅原因不再赘述。）   **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。**  基于1.中的算法思想与理解，对于鸢尾花数据集与动物园分类数据集使用Python实现BP算法并进行结果可视化，观察数据的分类情况与正确率随着迭代次数的变化。  （1）应用于鸢尾花数据集：   1. **def** train\_iris(): 2. data = np.array(pd.read\_csv('data/iris.csv')) 3. X = data[:, np.arange(1, 5)] 4. X = X.astype(np.float) 5. X = np.mat(X) 6. y = data[:, np.arange(5, 6)] 7. y[y == 'Iris-setosa'] = np.uint8(1) 8. y[y == 'Iris-versicolor'] = np.uint8(2) 9. y[y == 'Iris-virginica'] = np.uint8(3) 10. y = y.astype(np.uint8) 11. y = np.mat(y) 13. accus = [] 15. **for** i **in** range(0,100): 16. weights = tools.train(X, y, num\_layer=1, num\_nodes=25, regularization\_index=0.01, iters=i) 17. **def** readable\_predict(X, thetas): 18. lens = len(np.array(y)) 19. count = 0 20. **for** idx **in** range(0, lens - 1): 21. **print**('predict:', (np.argmax(tools.compute\_output(thetas,X[idx])[-1]) + 1)) 22. **print**('real tag:', np.array(y)[idx][0]) 23. **if** ((np.argmax(tools.compute\_output(thetas,X[idx])[-1]) + 1) == np.array(y)[idx][0]): 24. count += 1 25. acc = count / lens 26. accus.append(acc) 27. **print**("准确率为", acc) 28. readable\_predict(X, weights) 29. plt.rcParams['font.sans-serif'] = ['SimHei'] 30. plt.rcParams['axes.unicode\_minus'] = False 31. # 绘制准确率随着迭代次数的变化情况 32. plt.plot(range(len(accus)), accus) 33. plt.xlabel(u'迭代次数') 34. plt.ylabel(u'准确率') 35. plt.title("准确率随迭代次数的变化") 36. plt.show()   （2）应用于动物园分类数据集：   1. **def** train\_zoo(): 2. data = np.array(pd.read\_csv('data/zoo.csv')) 3. x\_ori = data[:, np.arange(1, 17)] 4. x\_ori = x\_ori.astype(np.float) 5. y\_ori = data[:, np.arange(17, 18)] 6. y\_ori = y\_ori.astype(np.uint8) 7. X = np.mat(x\_ori) 8. y = np.mat(y\_ori) 10. accus = [] 12. **for** i **in** range(0,500): 13. weights = tools.train(X, y, num\_layer=1, num\_nodes=25, regularization\_index=0.01, iters=i) 14. **def** readable\_predict(X, thetas): 15. lens = len(np.array(y)) 16. count = 0 17. **for** idx **in** range(0, lens - 1): 18. **print**('predict:', (np.argmax(tools.compute\_output(thetas,X[idx])[-1]) + 1)) 19. **print**('real tag:', np.array(y)[idx][0]) 20. **if** ((np.argmax(tools.compute\_output(thetas,X[idx])[-1]) + 1) == np.array(y)[idx][0]): 21. count += 1 22. acc = count / lens 23. accus.append(acc) 24. **print**("准确率为", acc) 25. readable\_predict(X, weights) 26. plt.rcParams['font.sans-serif'] = ['SimHei'] 27. plt.rcParams['axes.unicode\_minus'] = False 28. # 绘制准确率随着迭代次数的变化情况 29. plt.plot(range(len(accus)), accus) 30. plt.xlabel(u'迭代次数') 31. plt.ylabel(u'准确率') 32. plt.title("准确率随迭代次数的变化") 33. plt.show()   （3）sigmoid函数及其导数：   1. **import** numpy as np  4. # sigmoid 函数的定义 5. **def** sigmoid(x): 6. **return** 1 / (1 + np.exp(-x))  9. # sigmoid函数的导数，其中out是隐藏层的输出（激励），即sigmoid函数本身的输出。 10. # np.multiply函数的作用是矩阵对应的元素分别相乘。 11. **def** derivative\_sigmoid(out): 12. **return** np.multiply(out, (1-out))   （4）初始化权值矩阵   1. # 初始化权值矩阵，按照PPT中的方案初始化为-1/2到1/2中的随机值。 2. **def** set\_initial\_weights(num\_layer, num\_nodes, num\_input, num\_output): 3. # 隐藏层的节点个数 4. layerNum = [num\_nodes **for** i **in** range(num\_layer)] 5. # 各层的节点个数（分别是输出层 隐藏层 输出层） 6. layersNums = [num\_input] + layerNum + [num\_output] 7. # 代表各层权值的矩阵 8. Ws = [] 9. **for** idx, unit **in** enumerate(layersNums): 10. **if** idx == len(layersNums) - 1: 11. **break** 12. nextUnit = layersNums[idx + 1] 13. w = np.random.rand(nextUnit, unit + 1) - 0.5 14. Ws.append(w) 15. **return** Ws   （5）代价函数   1. # 带入报告中我推导出的代价函数，运算代价函数。 2. **def** compute\_cost(thetas, y, regularization\_index, a=None): 3. # 样本条目数 4. num\_record = y.shape[0] 5. # 训练完成后整个网络与真实值之差 6. # 用矩阵的转置进行乘法运算 7. error = -np.sum(np.multiply(y.T, np.log(a[-1])) + np.multiply((1 - y).T, np.log(1 - a[-1]))) 8. # 正则化项 9. reg = np.sum([np.sum(np.multiply(theta[:, 1:], theta[:, 1:])) **for** theta **in** thetas]) 10. # 代入推导出的J公式，返回即可。 11. **return** (1.0 / num\_record) \* error + (1.0 / (2 \* num\_record)) \* regularization\_index \* reg   （6）标签规范化   1. # 此函数的作用说明： 2. # 矩阵做运算时需要按标签类型的个数进行向量化，而输入数据只有一列。 3. # 此函数对y标签矩阵进行向量化 4. **def** vectorize\_labels(y): 5. # 使用np.ravel()可以影响原始矩阵 6. labels = set(np.ravel(y)) 7. num\_labels\_class = len(labels) 8. # 标签的最小值 9. mini\_label = min(labels) 10. # 向量化之后的标签矩阵 11. vector\_labels = np.zeros((y.shape[0], num\_labels\_class), np.float64) 12. **for** row, label **in** enumerate(y): 13. vector\_labels[row, label - mini\_label] = 1.0 14. **return** vector\_labels   （7）激励的计算   1. # 计算输出（激励值） 2. **def** compute\_output(weights, X): 3. # 层数 4. mat\_num\_layer = list(range(len(weights) + 1)) 5. num\_layer = len(mat\_num\_layer) 6. outputs = list(range(num\_layer)) 7. # 前向传播 8. **for** layer **in** mat\_num\_layer: 9. # 输入层 10. **if** layer == 0: 11. outputs[layer] = X.T 12. # 隐藏层与输出层 13. **else**: 14. z = weights[layer - 1] \* outputs[layer - 1] 15. outputs[layer] = sigmoid(z) 16. # 输入层与隐藏层添加偏置 17. **if** layer != num\_layer - 1: 18. outputs[layer] = np.concatenate((np.ones((1, outputs[layer].shape[1])), outputs[layer])) 19. **return** outputs   （8）反向传播与梯度计算   1. # 反向传播，计算梯度表达式 2. **def** gradient(thetas, a, y, regularization\_index): 3. num\_record = y.shape[0] 4. mat\_layers = list(range(len(thetas) + 1)) 5. num\_layers = len(mat\_layers) 6. d = list(range(len(mat\_layers))) 7. delta = [np.zeros(theta.shape) **for** theta **in** thetas] 8. **for** layer **in** mat\_layers[::-1]: 9. **if** layer == 0: 10. # 输入层不计算误差 11. **break** 12. **if** layer == num\_layers - 1: 13. # 输出层误差 14. d[layer] = a[layer] - y.T 15. **else**: 16. # 忽略偏置 17. d[layer] = np.multiply((thetas[layer][:, 1:].T \* d[layer + 1]), derivative\_sigmoid(a[layer][1:, :])) 19. **for** l **in** mat\_layers[0:num\_layers - 1]: 20. delta[l] += d[l + 1] \* (a[l].T) 21. D = [np.zeros(theta.shape) **for** theta **in** thetas] 22. **for** l **in** range(len(thetas)): 23. theta = thetas[l] 24. # 偏置更新增量 25. D[l][:, 0] = (1.0 / num\_record) \* (delta[l][0:, 0].reshape(1, -1)) 26. # 权值更新增量 27. D[l][:, 1:] = (1.0 / num\_record) \* (delta[l][0:, 1:] + regularization\_index \* theta[:, 1:]) 28. **return** D   （9）梯度下降与参数更新   1. # 使用梯度下降法更新权重矩阵 2. **def** gradientDescent(thetas, X, y, alpha, regularization\_index): 3. # 样本数，特征数 4. m, n = X.shape 5. # 前向传播计算各个神经元的激活值 6. a = compute\_output(thetas, X) 7. # 反向传播计算梯度增量 8. D = gradient(thetas, a, y, regularization\_index) 9. # 计算预测代价 10. J = compute\_cost(thetas,y,regularization\_index,a=a) 11. # 更新权值 12. **for** l **in** range(len(thetas)): 13. thetas[l] = thetas[l] - alpha \* D[l] 14. **if** np.isnan(J): 15. J = np.inf 16. **return** J, thetas   （10）模型训练   1. **def** train(X, y, num\_layer=1, num\_nodes=8, alpha=1, regularization\_index=0, iters=0): 2. # 记录条数、输入维度数 3. num\_records, num\_input = X.shape 4. y = vectorize\_labels(y) 5. classNum = y.shape[1] 6. weights = set\_initial\_weights(num\_input=num\_input, num\_layer=num\_layer, num\_nodes=num\_nodes, num\_output=classNum) 7. **for** i **in** range(iters): 8. error, weights = gradientDescent(weights, X, y, alpha=alpha, regularization\_index=regularization\_index) 9. **return** weights | | | | | | |
| 四、实验结果及分析  （1）动物园分类数据集分类结果  不使用正则化，设置迭代次数为50次，查看分类准确率    准确率仅为13.8%，说明如果不用正则化，则过拟合的问题相当严重。  使用正则化，设置迭代次数为20次，查看分类准确率    准确率为95%，过拟合问题大大改善。效果显著。  使用正则化，设置迭代次数为100次，查看分类准确率    准确率为99%  使用正则化，设置迭代次数为500次，查看分类准确率    准确率为99.1%，基本收敛。  使用正则化，画图，查看准确率随着迭代次数变化的变化情况    （2）鸢尾花数据集分类结果  不使用正则化，设置迭代次数为50次，查看分类准确率    准确率仅为73.3%  使用正则化，设置迭代次数为20次，查看分类准确率    准确率为88%  使用正则化，设置迭代次数为100次，查看分类准确率    准确率为90.6%  使用正则化，设置迭代次数为500次，查看分类准确率    准确率为94.6%  使用正则化，画图，查看准确率随着迭代次数变化的变化情况 | | | | | | |
| 五、实验总结与体会，亮点分析  总结体会：  本次实验中的核心思想是利用BP算法实现了简单的人工神经网络，进行分类问题的求解。神经网络的效果真的很好，特别是在加入正则化解决过拟合问题之后，分类的准确率相当之高。本次算法模型建立与编程实现的过程使我受益匪浅。  本次实验的亮点主要是：  进行了详细的可视化，代码结构简洁清楚，算法正确。 | | | | | | |