**《机器学习基础》实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | | **2022级计算机科学与技术(卓越) 01班** | | | **姓名** |  |
| **实验题目** | **BP算法实践** | | | | | |
| **实验时间** | **2024/04/21** | | **实验地点** | **DS3401** | | |
| **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。 | | | | | | |
| 1. 实验过程或算法（源程序）   1 .神经网络模型  实验一使用对数几率回归模型解决了特征数较少情况下的二分类问 题，本质上只能解决线性分类。当特征数增多，模型变得复杂，线性分类 就不再能够满足样本的分类需要。为了适应更加复杂的模型分类，有人在 神经元模型和感知机的基础上，提出了神经网路模型。  神经网络模型在输入层和输出层之间加入了隐藏层，对输入数据做了 空间变换，将其从线性不可分转换为近似的线性可分。根据输入数据的不 同复杂程度（本实验中即为特征数量），可以选择不同层数的隐藏层，来 更好地拟合复杂函数。如图为一个只含一层隐含层的神经网络模型：  IMG_256  假设样本的数据特征有 d 类，分类标签有 l 类，则构建一个含有 d 个神经 元输入层，l 个神经元输出层的前馈神经网络。为了使网络更好地拟合样 本，加入了一层含有 q 个神经元隐藏层（q 为超参数，可以人为设置）。 模型中输入层->隐藏层间的连接权值为 vih，隐藏层第 h 个神经元的阈 值为Yh。对于输入层的输入，隐藏层第 h 个神经元接收的输入为：  设激活函数为 f(x)，则隐藏层第 h 个神经元向输出层的输出为 bh=f(αh-ϒh)。 隐藏层的激活函数一般使用 Sigmoid 函数。  模型中隐藏层->输出层的连接权值为 whj，输出层第 j 个神经元的阈值 为θj。对于隐藏层的输入，输出层第 j 个神经元接收的输入为  设激活函数为 g(x)，则隐藏层第 h 个神经元向输出层的输出为 yj=g(βj-θj)。 对于多分类问题，输出层的激活函数一般使用 Softmax 函数。但由于 Softmax 激活在反向传播时导数不连续，导致损失函数上下波动，有可能不收敛。因此还是使用 Sigmoid 函数，取预测结果的最大值作为最终的分类结果，仍然可以较好的实现多分类。  每经过一次前向传播，可以得到模型的损失函数：  2 .BP算法原理  前馈神经网络直接的连接权重和神经元的阈值并没有给定，需要根据 训练样本训练得到。通过逆向传播算法迭代学习参数是比较成功的训练多 层前馈神经网络的学习方法。  逆向传播算法就是使用梯度下降法，通过对损失函数求导，反向更新参数的过程。每次迭代对参数的调整如下：  **实验代码：**   1. import time 2. import numpy as np 3. import pandas as pd 4. import random 5. import matplotlib 6. import matplotlib.pyplot as plt 7. from logging import error 8. matplotlib.rcParams['font.sans-serif'] = ['KaiTi'] 9. def sigmoid(X): 10. return 1 / (1 + np.exp(-X)) 11. def sigmoid\_diff(y): 12. return y \* (1 - y) 13. def tanh(X): 14. return (np.exp(X) - np.exp(-X)) / (np.exp(X) + np.exp(-X)) 15. def tanh\_diff(y): 16. return 1 - y \*\* 2 17. def squared\_loss(y\_true, y\_pred): 18. return ((y\_true - y\_pred) \*\* 2).mean(axis=0).sum() / 2 19. def softmax(X): 20. 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* 21. def one\_hot\_encoder(y, class\_encoder=None): 22. if class\_encoder == None: 23. y\_set = set(y.ravel()) 24. class\_encoder = {label: idx for idx, label in enumerate(y\_set)} 25. n\_classes = len(class\_encoder) 26. n\_samples = len(y) 27. y\_one\_hot = np.zeros((n\_samples, n\_classes), dtype=int) + 0.01 28. for idx, label in enumerate(y.ravel()): 29. y\_one\_hot[idx, class\_encoder[label]] = 1 - 0.01 30. return y\_one\_hot 31. def one\_hot\_decoder(y\_one\_hot, class\_decoder=None): 32. if class\_decoder == None: 33. class\_decoder = {label: idx for idx, label in enumerate(range(y\_one\_hot.shape[1]))} 34. y\_transfer = y\_one\_hot.copy() 35. for idx, col in enumerate(y\_transfer.T): 36. *# 注意，这里的col只是 y\_transfer 的一个视图* 37. col[col == 1] = class\_decoder[idx] 38. y = np.max(y\_transfer, axis=1).astype(int) 39. return y.reshape(-1, 1)  *# [r,1]* 40. ACTIVATIONS = {"sigmoid": sigmoid, "tanh": tanh} 41. DIFF = {"sigmoid": sigmoid\_diff, "tanh": tanh\_diff} 42. class BpNet: 43. def \_\_init\_\_(self, hidden\_layer\_sizes=(6,), max\_iter=1000, learning\_rate=0.1, epsilon=1e-8, activation="sigmoid",batch=None) -> None: 44. ''' 45. BpNet初始化 46. @param hidden\_layer\_sizes:自定义隐含层元组 47. @param max\_iter:最大迭代次数 48. @param learning\_rate:学习率 49. @param epsilon:最大误差精度 50. @param activation:激活函数 51. @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； 52. ''' 53. self.hidden\_layer\_sizes = list(hidden\_layer\_sizes) 54. self.max\_iter = max\_iter 55. self.activation = activation 56. self.learning\_rate = learning\_rate 57. self.epsilon = epsilon 58. self.batch = batch 59. def fit(self, X, y): 60. ''' 训练数据 ''' 61. *# X的预处理* 62. X = self.fit\_transform(X) 63. *# Y的维数判断* 64. if y.ndim == 1: 65. y = y.reshape((-1, 1)) 66. *# 转为独热编码* 67. if y.shape[1] == 1: 68. y\_set = set(y.ravel()) 69. *# 经过测验，one\_hot\_encoder 和 one\_hot\_decoder基本没有问题* 70. self.class\_encoder = {label: idx for idx, label in enumerate(y\_set)} 71. self.class\_decoder = {idx: label for idx, label in enumerate(y\_set)} 72. y = one\_hot\_encoder(y, self.class\_encoder) 73. *# 层数设置* 74. n\_samples, n\_features = X.shape 75. n\_output = y.shape[1] 76. layer\_units = ([n\_features] + self.hidden\_layer\_sizes + [n\_output]) 77. self.layer\_units = layer\_units 78. self.n\_layers\_ = len(layer\_units) 79. *# 初始化 每一层的权重和阈值* 80. self.coef\_ = [] 81. self.intercept\_ = [] 82. for i in range(self.n\_layers\_ - 1): 83. coef\_init = np.random.random((layer\_units[i], layer\_units[i + 1])) 84. intercept\_init = np.random.random(layer\_units[i + 1]) 85. self.coef\_.append(coef\_init) 86. self.intercept\_.append(intercept\_init) 87. *# 每一层的 被激活的单元值* 88. activations = [X] + [None] \* (len(layer\_units) - 1) 89. deltas = [None] \* (len(activations) - 1) 90. *# 限定梯度计算* 91. coef\_grads = [np.empty((n\_in, n\_out)) for n\_in, n\_out in zip(layer\_units[:-1], layer\_units[1:])] 92. intercept\_grads = [np.empty(n\_out) for n\_out in layer\_units[1:]] 93. *# 记录迭代次数* 94. self.n\_iter\_ = 0 95. *# 记录迭代样本数和损失函数* 96. self.loss\_curve\_ = [] 97. *# 计算batch* 98. if self.batch is None or self.batch > n\_samples: 99. self.batch = n\_samples 100. elif self.batch <= 0: 101. self.batch = 1 102. *# 获取训练中需要的idx\_batch\_list* 103. idx\_batch\_list = [] 104. last\_idx = 0 105. for idx in range(self.batch, n\_samples + self.batch, self.batch): 106. idx = min(idx, n\_samples) 107. idx\_batch\_list.append([last\_idx, idx]) 108. last\_idx = idx 109. *# print("训练结构：",idx\_batch\_list)* 110. self.\_fit(X, y, activations, deltas, coef\_grads, intercept\_grads, layer\_units, idx\_batch\_list) 111. def predict\_proba(self, X): 112. ''' 预测概率 ''' 113. if X.shape[1] != self.layer\_units[0]: 114. error("输入的X", {X.shape}, "维数不正确") 115. return False 116. *# X要归一化* 117. X = self.transform(X) 118. *# 初始化神经网络层，确定 activations 每一个维度的大小* 119. activations = [X] 120. for i in range(1, self.n\_layers\_): 121. activations.append(np.empty((X.shape[0], self.layer\_units[i]))) 122. *# 前向传播，计算* 123. activations = self.\_forwardprop(activations) 124. y\_prob = activations[-1] 125. return y\_prob 126. def predict(self, X):  *# (n\_samples,n\_features)* 127. ''' 预测 ''' 128. y\_prob = self.predict\_proba(X) 129. y\_one\_hot = np.zeros(y\_prob.shape) 130. y\_max = np.argmax(y\_prob, axis=1) 131. for ridx, midx in enumerate(y\_max): 132. y\_one\_hot[ridx, midx] = 1 133. return one\_hot\_decoder(y\_one\_hot, self.class\_decoder) 134. def fit\_transform(self, X): 135. ''' 记录 + X的预处理：归一化 ''' 136. self.x\_max = np.max(X, axis=0) 137. self.x\_min = np.min(X, axis=0) 138. 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* 139. return X 140. def transform(self, X): 141. ''' X的预处理：归一化 ''' 142. 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* 143. return X 144. def score(self, X, y): 145. ''' 预测准确率 ''' 146. *# Y的预处理* 147. if y.ndim == 1: 148. y = y.reshape((-1, 1)) 149. y\_pred = self.predict(X) 150. return (y == y\_pred).mean() 151. def \_fit(self, X, y, activations, deltas, coef\_grads, intercept\_grads, layer\_units, idx\_batch\_list, shuffle=False): 152. n\_samples = len(X) 153. n\_bp\_cnt = len(idx\_batch\_list) 154. sample\_idx = np.arange(n\_samples, dtype=int) 155. *# 开始迭代* 156. for it in range(self.max\_iter): 157. accumulated\_loss = 0.0 158. if shuffle: 159. random.shuffle(sample\_idx) 160. for idx\_batch in idx\_batch\_list: 161. *# 获取训练样本* 162. li,ri = idx\_batch 163. sampleX, sampley = X[li:ri], y[li:ri] 164. *# 前向传播，计算预测值* 165. activations[0]= sampleX 166. activations = self.\_forwardprop(activations) 167. *# 计算均方误差* 168. accumulated\_loss += squared\_loss(sampley, activations[-1]) 169. *# 反向传播，计算梯度，更新权值和阈值* 170. self.\_backprop(sampley, activations, deltas, coef\_grads, intercept\_grads) 171. self.n\_iter\_ += 1 172. self.loss\_curve\_.append(accumulated\_loss / n\_bp\_cnt) 173. if self.loss\_curve\_[-1] < self.epsilon: 174. break 175. def \_forwardprop(self, activations): 176. activation\_fun = ACTIVATIONS[self.activation] 177. *# 逐层回归+激活* 178. for i in range(self.n\_layers\_ - 1): 179. activations[i + 1] = np.dot(activations[i], self.coef\_[i]) + self.intercept\_[i] 180. activations[i + 1] = activation\_fun(activations[i + 1]) 181. *# TODO 数据前推，尚未进行softmax处理* 182. *# activations[i + 1] = softmax(activations[i + 1])* 183. return activations 184. def \_backprop(self, y, activations, deltas, coef\_grads, intercept\_grads): 185. diff\_fun = DIFF[self.activation] 186. n\_samples = len(y) 187. *# 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* 188. last = self.n\_layers\_ - 2 189. *# TODO 反向传播，尚未进行softmax处理* 190. *# 输出层->隐含层* 191. deltas[last] = (activations[-1] - y) \* diff\_fun(activations[-1]) 192. coef\_grads, intercept\_grads = self.\_cal\_loss\_grad(last, n\_samples, activations, deltas, coef\_grads, 193. intercept\_grads) 194. *# 隐含层->隐含层* 195. for i in range(last, 0, -1): 196. deltas[i - 1] = np.dot(deltas[i], self.coef\_[i].T) \* diff\_fun(activations[i]) 197. coef\_grads, intercept\_grads = self.\_cal\_loss\_grad(i - 1, n\_samples, activations, deltas, coef\_grads, 198. intercept\_grads) 199. *# TODO 学习器优化* 200. for i in range(self.n\_layers\_ - 1): 201. self.coef\_[i] += -1 \* self.learning\_rate \* coef\_grads[i] 202. self.intercept\_[i] += -1 \* self.learning\_rate \* intercept\_grads[i] 203. return None 204. def \_cal\_loss\_grad(self, layer, n\_samples, activations, deltas, coef\_grads, intercept\_grads): 205. *# deltas : 右层节点对应残差* 206. coef\_grads[layer] = np.dot(activations[layer].T, deltas[layer]) 207. *# coef\_grads[layer] /= n\_samples* 208. intercept\_grads[layer] = np.mean(deltas[layer], axis=0) 209. return coef\_grads, intercept\_grads 210. def get\_params(self): 211. ''' 获取模型参数 ''' 212. return self.coef\_, self.intercept\_ 213. def train\_test\_split(X, Y, train\_percent=0.7, shuffle=True, seed=None): 214. ''' 自定义数据分割 ''' 215. n\_smaples = X.shape[0] 216. if shuffle: 217. idx = np.arange(n\_smaples, dtype=int) 218. if seed: 219. random.seed(2) 220. random.shuffle(idx) 221. X = X[idx] 222. Y = Y[idx] 223. n\_train = int(np.floor(n\_smaples \* train\_percent)) 224. trainX, testX = X[0:n\_train], X[n\_train:-1] 225. trainY, testY = Y[0:n\_train], Y[n\_train:-1] 226. return trainX, testX, trainY, testY 227. def get\_iris\_data(filepath='./iris.csv'): 228. ''' 获取鸢尾花数据集 ''' 229. iris\_df = pd.read\_csv(filepath) 230. iris\_data = iris\_df.values 231. X = iris\_data[:, :-1] 232. Y = iris\_data[:, -1][:, np.newaxis] 233. return X, Y 234. def get\_wine\_data(filepath='./wine.data'): 235. ''' 获取红酒数据集 ''' 236. wine\_data = np.loadtxt(filepath, delimiter=",") 237. Y = wine\_data[:, 0][:, np.newaxis] 238. X = wine\_data[:, 1:] 239. return X, Y 240. def get\_digits\_data(): 241. ''' 获取手写数字数据集 ''' 242. from sklearn.datasets import load\_digits 243. dig = load\_digits() 244. X = dig.data 245. Y = dig.target 246. return X, Y 247. def test(data\_name): 248. ''' 数据集统一测试 ''' 249. my\_dataset = { 250. "iris": { 251. "name": "鸢尾花数据集", 252. "get\_fun": get\_iris\_data, 253. "bp": BpNet(hidden\_layer\_sizes=(6, 4), max\_iter=10000) 254. }, 255. "wine": { 256. "name": "红酒数据集", 257. "get\_fun": get\_wine\_data, 258. "bp": BpNet(hidden\_layer\_sizes=(8, 6,), max\_iter=1000,batch=-1) 259. } 260. } 261. print("=" \* 30, my\_dataset[data\_name]["name"], "=" \* 30) 262. X, Y = my\_dataset[data\_name]["get\_fun"]() 263. trainX, testX, trainY, testY = train\_test\_split(X, Y, seed=1) 264. bp = my\_dataset[data\_name]["bp"]  *# type: BpNet* 265. start\_time = time.time() 266. print("模型开始训练") 267. bp.fit(trainX, trainY) 268. print("模型结构：", bp.layer\_units) 269. print("模型训练结束，用时%.3fs" % ((time.time() - start\_time) / 60)) 270. *# 绘制训练过程* 271. plt.plot(range(len(bp.loss\_curve\_)), bp.loss\_curve\_) 272. plt.title(my\_dataset[data\_name]["name"] + "训练过程记录") 273. plt.legend(['loss']) 274. plt.xlabel("iter") 275. plt.ylabel("loss") 276. plt.show() 277. *# 预测和评估* 278. *# print("真实值：",testY.ravel())* 279. *# print("预测值：",bp.predict(testX).ravel())* 280. print("测试集：") 281. predY = bp.predict(testX) 282. print("损失函数值：%.3f" % (squared\_loss(testY, predY))) 283. print("预测准确率：%.3f" % (bp.score(testX, testY))) 284. if \_\_name\_\_ == "\_\_main\_\_": 285. test("iris") 286. test("wine") | | | | | | |
| 四、实验结果及分析  BP算法的核心思想：使用梯度下降来搜索可能的权向量的假设空间，以找到最佳的拟合样例的权向量。具体而言，即利用损失函数，每次向损失函数负梯度方向移动，直到损失函数取得最小值。  或者说，反向传播算法，是根据损失函数，求出损失函数关于每一层的权值及偏置项的偏导数，也称为梯度，用该值更新初始的权值和偏置项，一直更新到损失函数取得最小值或是设置的迭代次数完成为止。以此来计算神经网络中的最佳的参数。  实验结果：  ======================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0.022s  测试集：  损失函数值：0.023  预测准确率：0.955    ======================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0.297s  测试集：  损失函数值：0.019  预测准确率：0.962    **分析：**  对于训练过程中loss骤增之后又骤减，通过查阅资料分析了以下可能造成原因：  1.学习率设置不当：  如果学习率过高，可能会导致网络在损失空间中跳跃，从而导致损失值的突然增加。随后，可能会降低学习率以减少波动。  如果学习率适中，损失值在一段时间内会逐渐减少。  如果学习率过低，损失值可能会很慢地减少。这可能需要更长的时间。  2.梯度消失或梯度爆炸：  梯度消失或梯度爆炸可能导致损失值的骤增，然后通过使用梯度裁剪或者使用合适的激活函数等方式得到解决，损失值会再次下降。  通过调整learning rate和迭代次数，该问题得到有效缓解。（下图所示） | | | | | | |