机器学习第十三次作业

5.3

$$
\begin{aligned}
&\Delta v_{ih} \\
&= -\eta \frac{\partial E_K}{\partial v_{ih}} \\
&= -\eta \Sigma_{j=1}^{l} \frac{\partial E_K}{\partial \hat{y}_j^k} \frac{\partial \hat{y}_j^k}{\partial \beta_j} \frac{\partial \beta_j}{\partial b_h} \frac{\partial b_h}{\partial \alpha_h} \frac{\partial \alpha_h}{\partial v_{ih}} \\
&= -\eta \Sigma_{j=1}^{l} \frac{\partial E_K}{\partial \hat{y}_j^k} \frac{\partial \hat{y}_j^k}{\partial \beta_j} \frac{\partial \beta_j}{\partial b_h} f'(\alpha_h - \gamma_h) x_i \\
&= -\eta \Sigma_{j=1}^{l} \frac{\partial E_K}{\partial \hat{y}_j^k} \frac{\partial \hat{y}_j^k}{\partial \beta_j} w_{hj} f'(\alpha_h - \gamma_h) x_i \\
&= -\eta \Sigma_{j=1}^{l} g_j w_{hj} f'(\alpha_h - \gamma_h) x_i \\
&= \eta b_h (1 - b_h) \Sigma_{j=1}^{l} g_j w_{hj} x_i \\
&= \eta e_h x_i
\end{aligned}
$$

5.5

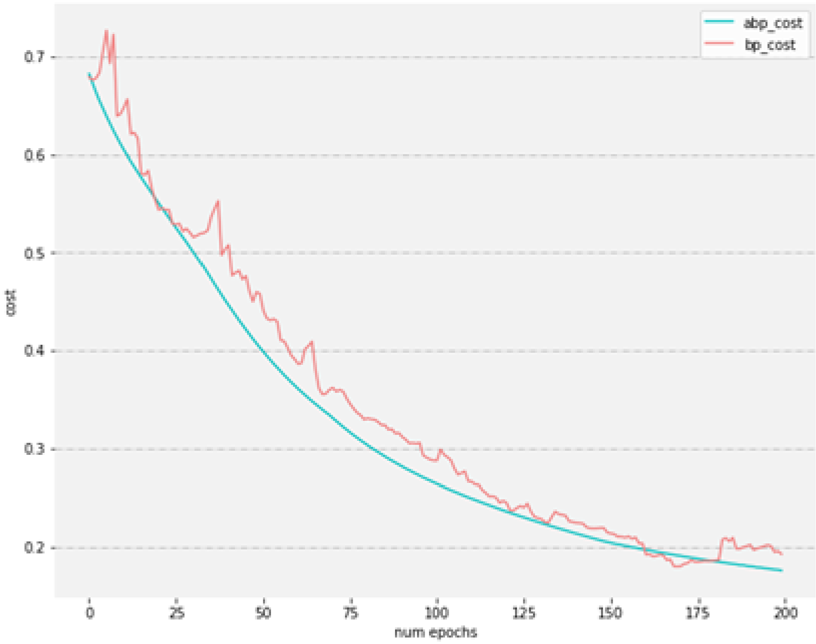
```
# 模块调用声明
import numpy as np
import pandas as pd
import sys

from scipy.special import expit
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

import matplotlib.pyplot as plt

melon_dataset = pd.read_excel('C:/Users/mi/Desktop/melon.xlsx')

# 将字符变量化为整数
features = ['色泽','根蒂','敲声','纹理','脐部','触感','好瓜']
for feature in features:
    try:
        melon_dataset[feature] =
LabelEncoder().fit_transform(melon_dataset[feature].apply(int))
    except:
        melon_dataset[feature] =
LabelEncoder().fit_transform(melon_dataset[feature])

# 划分数据集
X = melon_dataset.iloc[:,1:-1]
y = melon_dataset['好瓜']

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
test_size=0.25)



# 定义单隐层感知器
```

```python
class MLNN(object):
    # 初始构造函数
    def 
__init__(self,n_output,n_features,n_hidden=30,l1=0.0,l2=0.0,epochs=500,eta=0.001
,alpha=0.0,decrease_const=0.0,shuffle=True,minibatches=1,random_state=None):
        np.random.seed(random_state)
        # 输出单元数量
        self.n_output = n_output
        # 输入单元数量
        self.n_features = n_features
        # 隐层单元数量
        self.n_hidden = n_hidden
        # 输入-隐层-输出的路径权重
        self.w1, self.w2 = self._initialize_weights()
        # L1正则化参数lambda
        self.l1 = l1
        # L2正则化参数lambda
        self.l2 = l2
        # 迭代次数
        self.epochs = epochs
        # 学习率
        self.eta = eta
        # 动量学习进度参数，用于加快权重更新的学习
        self.alpha = alpha
        # 用于降低自适应学习速率n的常数d，随迭代次数的增加而递减以保证收敛
        self.decrease_const = decrease_const
        # 在每次迭代前打乱训练集的顺序，防止算法陷入死循环
        self.shuffle = shuffle
        # 每次迭代中将训练数据划分为k个小的批次，为加快学习过程，梯度由各个批次分别计算
        self.minibatches = minibatches
        self.cost_ = []

    # 类别变量化为整数
    def _encode_labels(self,y,k):
        onehot = np.zeros((k,y.shape[0]))
        for idx, val in enumerate(y):
            onehot[val,idx] = 1.0
        return onehot

    # 初始化权值
    def _initialize_weights(self):
        # 输入层-隐层
        w1 = np.random.uniform(-1.0,1.0,size=self.n_hidden*(self.n_features + 
1))
        # 最后一列为阈值
        w1 = w1.reshape(self.n_hidden, self.n_features + 1)
        # 隐层-输出层
        w2 = np.random.uniform(-1.0,1.0,size=self.n_output*(self.n_hidden + 1))
        w2 = w2.reshape(self.n_output, self.n_hidden + 1)
        return w1,w2

    # Sigmoid函数
    def _sigmoid(self,z):
        # expit(z) = 1.0/(1.0 + np.exp(-z))
        return expit(z)

    # 计算Sigmoid函数的导数
    def _sigmoid_gradient(self,z):
```

```python
        sg = self._sigmoid(z)
        return sg * (1 - sg)

    # 添加偏置项（初始化为1）
    def _add_bias_unit(self,X,how='column'):
        if how == 'column':
            X_new = np.ones((X.shape[0],X.shape[1]+1))
            X_new[:,1:] = X
        elif how == 'row':
            X_new = np.ones((X.shape[0]+1,X.shape[1]))
            X_new[1:,:] = X
        else:
            raise AttributeError("'how' must be 'column' or 'row'")
        return X_new

    # 前向传播
    def _feedforward(self,X,w1,w2):
        # 输入层原始数据
        a1 = self._add_bias_unit(X, how = 'column')
        # 加权数据1
        z2 = w1.dot(a1.T)
        # 作用激活函数
        a2 = self._sigmoid(z2)
        # 隐层输出数据
        a2 = self._add_bias_unit(a2, how = 'row')
        # 加权数据2
        z3 = w2.dot(a2)
        # 作用激活函数
        a3 = self._sigmoid(z3)
        return a1,z2,a2,z3,a3

    # L1正则化(矩阵1范数)
    def _L1_reg(self,lambda_,w1,w2):
        return (lambda_/2.0)*(np.abs(w1[:,1:]).sum() + np.abs(w2[:,1:]).sum())

    # L2正则化(矩阵2范数)
    def _L2_reg(self,lambda_,w1,w2):
        return (lambda_/2.0)*(np.abs(w1[:,1:]**2) + np.sum(w2[:,1:]**2))

    # 计算损失函数
    def _get_cost(self,y_enc,output,w1,w2):
        term1 = -y_enc * (np.log(output))
        term2 = (1 - y_enc) * np.log(output - 1)
        cost = np.sum(term1 - term2)
        L1_term = self._L1_reg(self.l1,w1,w2)
        L2_term = self._L2_reg(self.l2,w1,w2)
        cost = cost + L1_term + L2_term
        return cost

    # 误差反向传播
    def _get_gradient(self,a1,a2,a3,z2,y_enc,w1,w2):
        # 预测误差
        sigma3 = a3 - y_enc
        # 加权数据1
        z2 = self._add_bias_unit(z2,how='row')
        sigma2 = w2.T.dot(sigma3)*self._sigmoid_gradient(z2)
        sigma2 = sigma2[1:,:]
```

```
        grad1 = sigma2.dot(a1)
        grad2 = sigma3.dot(a2.T)
        # 正则化
        grad1[:,1:] += (w1[:,1:]*(self.l1 + self.l2))
        grad2[:,1:] += (w2[:,1:]*(self.l1 + self.l2))

        return grad1,grad2

    def predict(self,X):
        a1,z2,a2,z3,z3 = self._feedforward(X,self.w1,self.w2)
        y_pred = np.argmax(z3,axis=0)
        return y_pred

    def fit(self,X,y,print_progress=False):
        self.const_ = []
        X_data, y_data = X.copy(), y.copy()
        y_enc = self._encode_labels(y, self.n_output)

        delta_w1_prev = np.zeros(self.w1.shape)
        delta_w2_prev = np.zeros(self.w2.shape)

        for i in range(self.epochs):
            self.eta /= (1 + self.decrease_const*i)

            if print_progress:
                sys.stderr.write('\rEpoch: %d/%d' % (i + 1,self.epochs))
                sys.stderr.flush()
            if self.shuffle:
                idx = np.random.permutation(y_data.shape[0])
                X_data, y_data = X_data.iloc[idx], y_data.iloc[idx]

            mini = np.array_split(range(y_data.shape[0]),self.minibatches)

            for idx in mini:
                a1, z2, a2, z3, a3 =
self._feedforward(X.iloc[idx,:],self.w1,self.w2)
                cost =
self._get_cost(y_enc=y_enc[:,idx],output=3,w1=self.w1,w2=self.w2)
                self.cost_.append(cost)
                # 计算梯度
                grad1, grad2 =
self._get_gradient(a1=a1,a2=a2,a3=a3,z2=z2,y_enc=y_enc[:,idx],w1=self.w1,w2=self
.w2)
                # 更新权重
                delta_w1, delta_w2 = self.eta * grad1, self.eta * grad2

                self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
                self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
                delta_w1_prev, delta_w2_prev = delta_w1, delta_w2

        return self


nn = MLNN(n_output =
2,n_features=X_train.shape[1],n_hidden=50,l1=0.0,l2=0.0,epochs=1000,eta=0.001,al
pha=0.001,decrease_const=0.00001,shuffle=False,minibatches=1,random_state=1)
nn.fit(X_train,y_train,print_progress=True)
```

```
y_pred = nn.predict(X_test)
acc = np.sum(y_test == y_pred, axis=0)/ X_test.shape[0]
print('\n')
print(acc)
```



fgl数据

1. 模块调用说明

```
import pandas as pd
import numpy as np

from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression

import matplotlib.pyplot as plt
```

2. 数据处理

glass_data

Index	Id number	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	Type
0	1	1.52101	13.64	4.49	1.1	71.78	0.06	8.75	0	0	1
1	2	1.51761	13.89	3.6	1.36	72.73	0.48	7.83	0	0	1
2	3	1.51618	13.53	3.55	1.54	72.99	0.39	7.78	0	0	1
3	4	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0	0	1
4	5	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0	0	1
5	6	1.51596	12.79	3.61	1.62	72.97	0.64	8.07	0	0.26	1
6	7	1.51743	13.3	3.6	1.14	73.09	0.58	8.17	0	0	1
7	8	1.51756	13.15	3.61	1.05	73.24	0.57	8.24	0	0	1
8	9	1.51918	14.04	3.58	1.37	72.08	0.56	8.3	0	0	1
9	10	1.51755	13	3.6	1.36	72.99	0.57	8.4	0	0.11	1
10	11	1.51571	12.72	3.46	1.56	73.2	0.67	8.09	0	0.24	1

```
# 读入数据
glass_data = pd.read_excel('C:/Users/mi/Desktop/glass.xlsx')
X_raw = glass_data.iloc[:,1:10]
# 对X做标准化处理
X_scaled = preprocessing.scale(X_raw)
y = glass_data['Type']
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, stratify=y,
test_size=0.25)
```

3. 训练多批模型

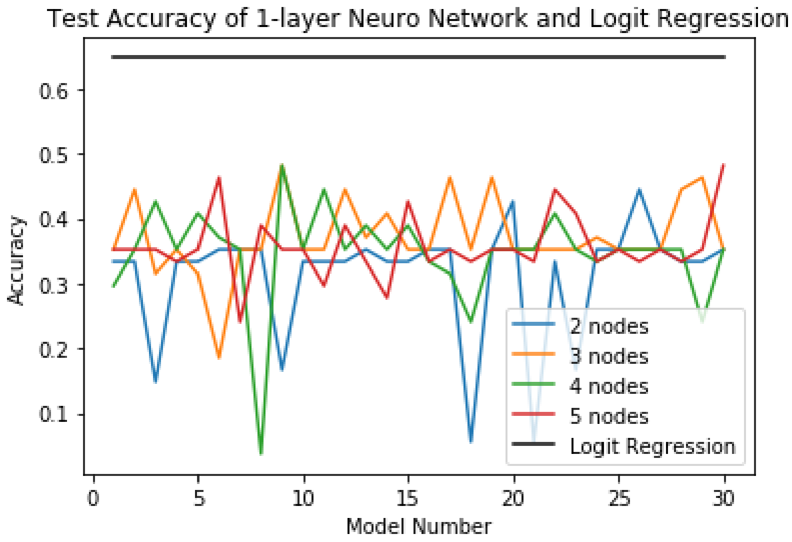
```
def neuro_acc(times,nodes):
    '''
    :params times: 模型重复训练次数
    :return accArray: 多次测试得到的得分
    '''
    accArr = []
    for i in range(times):
        # 单批训练
        #neuron = MLPClassifier(random_state=0,hidden_layer_sizes=
(1,nodes),solver='sgd')
        # 多批训练
        neuron = MLPClassifier(hidden_layer_sizes=(1,nodes),solver='sgd')
        neuron.fit(X_train,y_train)
        accArr.append(neuron.score(X_test,y_test))
    return accArr

def logi_acc(times):
    '''
    :params times: 模型重复训练次数
    :return accArray: 多次测试得到的得分
    '''
    accArr = []
    for i in range(times):
        logi = LogisticRegression()
        logi.fit(X_train,y_train)
        accArr.append(logi.score(X_test,y_test))
    return accArr
```

4. 训练误差

```
accArr2 = neuro_acc(30,2)
accArr3 = neuro_acc(30,3)
accArr4 = neuro_acc(30,4)
accArr5 = neuro_acc(30,5)

accArrlogi = logi_acc(30)
times = [x+1 for x in range(30)]


fig, ax = plt.subplots()
ax.plot(times,accArr2,label='2 nodes')
ax.plot(times,accArr3,label='3 nodes')
ax.plot(times,accArr4,label='4 nodes')
ax.plot(times,accArr5,label='5 nodes')
ax.plot(times,accArrlogi,color='black',label='Logit Regression')
```

```
ax.set_xlabel('Model Number')
ax.set_ylabel('Accuracy')
ax.set_title('Test Accuracy of 1-layer Neuro Network and Logit Regression')
ax.legend()
```

多批模型训练得分



单批模型训练得分