**机器学习应用实践（实验七） —MLP网络和BP算法**

**工业智能2201班 刘天行 20225354**

1. **实验目的**

1、 掌握 Scikit-Learn 中提供的 MLP 网络算法的使用。

2、 掌握 MLP 网络和 BP 算法的原理，并具备使用 python 实现 BP 神经网络的编程能力。

1. **开发环境：Visual Studio Code + Jupyter notebook拓展 + python 3.12.2**

文本

描述已自动生成

1. **实验内容**

1、 题目一（选做） ：采用 scikit-learn 中的 MLPClassifier 对红酒数据集进行分类，并通过特征和边界的可视化， 直观体会多层感知机网络中的隐层上神经元数量、隐层层数、激活函数、正则化项系数等超参数对模型复杂程度的影响。

具体内容：

（1）选取前两个特征，建立多层感知机网络进行多分类。

（2）可视化：通过散点图可视化数据样本（之前选择的两个特征） ，并画出模型训练后得到的决策边界。

【讨论一】 改变单隐层中神经元个数（如 10 个， 100 个） ， 其他参数不变， 观察其对决策边界的影响。

【讨论二】 改变神经网络深度（如深度为 2，每层 10 个神经元） ， 其他参数不变， 与讨论一进行对比， 观察神经网络深度其对决策边界的影响。

【讨论三】 在讨论一（或讨论二） 的基础上， 改变激活函数（如 tanh、 relu） ， 与讨论一（或讨论二）进行对比， 观察不同激活函数对决策边界的影响。

【讨论四】在讨论三的基础上，增大正则化系数，观察正则化对决策边界的影响。

总结：综合上述讨论，隐层上神经元数量、隐层层数、激活函数、正则化项系数对模型复杂程度有何影响。

2、题目二： 采用 scikit-learn 中的 MLPClassifier 对自带手写数字数据集进行分类

【讨论五】 结合模型复杂度与模型泛化误差之间的关系，调节模型超参数，提升模型泛化性能。可尝试调节隐层神经元个数和隐层数、激活函数、学习率、 正则项系数等超参数。

3、 题目三： 编写 BPNN 算法， 对 iris 数据集/手写数字集进行二分类或多分类。

1. **实验情况**

题目一：采用 scikit-learn 中的 MLPClassifier 对红酒数据集进行分类，并通过特征和边界的可视化， 直观体会多层感知机网络中的隐层上神经元数量、隐层层数、激活函数、正则化项系数等超参数对模型复杂程度的影响。

from sklearn.neural\_network import MLPClassifier

from sklearn.datasets import load\_wine

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

import numpy as np

from ipywidgets import interactive

from IPython.display import display

X, y = load\_wine(return\_X\_y=True)

X = X[:, :2]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.15, random\_state=520)

plt.rcParams['font.family'] = 'SimHei'      # 中文正常显示

plt.rcParams['axes.unicode\_minus'] = False  # 符号正常显示

# 定义函数来绘制决策边界，cmap 颜色可选 'coolwarm' 'spring' 'summer' 'autumn'

# 'winter' 'bone' 'copper' 'gray' 'pink' 'hot' 'rainbow' 'viridis'

def plot\_decision\_boundary(X, y, model, ax, steps=1000, cmap='coolwarm',title='决策边界'):

    """

    绘制决策边界, X 为数据集，y 为标签，model 为训练好的模型，ax 为绘制的坐标轴，steps 为网格的步长，cmap 为颜色

    """

    cmap = plt.get\_cmap(cmap)

    # 定义坐标轴范围,并创建网格

    x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

    y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, steps),

                         np.linspace(y\_min, y\_max, steps))

    # 计算 Z，即每个网格点的预测值

    Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)

    # 绘制等高线和训练集散点图

    ax.contourf(xx, yy, Z, cmap=cmap, alpha=0.8)

    scatter = ax.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap, edgecolor='k')

    # 图例

    legend1 = ax.legend(\*scatter.legend\_elements(),

                        title="Classes")

    ax.add\_artist(legend1)

        # 设置坐标轴范围和标题

    ax.set\_xlim(X[:,0].min()-0.5, X[:,0].max()+0.5)

    ax.set\_ylim(X[:,1].min()-0.5, X[:,1].max()+0.5)

    ax.set\_xticks(())

    ax.set\_yticks(())

    ax.set\_title(title)

clf = MLPClassifier(hidden\_layer\_sizes=np.intp(np.ones(5)\*10),

                    random\_state=1, max\_iter=10000).fit(X\_train, y\_train)

fig, ax = plt.subplots(figsize=(8,6))

plot\_decision\_boundary(X\_train, y\_train, clf, ax)

pre\_train = clf.predict(X\_train)

pre\_test = clf.predict(X\_test)

plt.title("训练集决策边界 train acc={}/{} test acc={}/{}"

        .format((pre\_train==y\_train).sum(),y\_train.size,

                (pre\_test==y\_test).sum(),y\_test.size))

plt.show()

图表, 散点图

描述已自动生成

【讨论一】 改变单隐层中神经元个数（如 10 个， 100 个） ， 其他参数不变， 观察其对决策边界的影响。

def change\_neuron\_num(n=10,ax = None):

    clf = MLPClassifier(hidden\_layer\_sizes=n,random\_state=520, max\_iter=10000, ).fit(X\_train, y\_train)

    # fig, ax = plt.subplots(figsize=(6,4))

    plot\_decision\_boundary(X\_train, y\_train, clf, ax = ax)

    # plot\_decision\_boundary(clf,X\_train,y\_train)

    pre\_train = clf.predict(X\_train)

    pre\_test = clf.predict(X\_test)

    plt.title("neuron num = {},train acc={}/{} test acc={}/{}"

            .format(n,(pre\_train==y\_train).sum(),y\_train.size,

                      (pre\_test==y\_test).sum(),y\_test.size),fontdict={'fontsize': 12})

    plt.tight\_layout()  # 调整图形边界

    plt.savefig("n={}.jpg".format(n), bbox\_inches='tight')

    # bbox\_inches='tight'参数使保存图形时，只包含紧凑的边界部分，避免保存过多白边

# interact(change\_neuron\_num, n=[5,20,35,50,75,100,150])

# talking\_neuron\_num = [5,20,35,50,75,100,150]

# for a\_neuron in talking\_neuron\_num:

#     change\_neuron\_num(a\_neuron)

plt.figure(figsize=[13, 4])

ax1 = plt.subplot(131)

change\_neuron\_num(5,ax1)

ax2 = plt.subplot(132)

change\_neuron\_num(20,ax2)

ax3 = plt.subplot(133)

change\_neuron\_num(35,ax3)

plt.figure(figsize=[13, 4])

ax4 = plt.subplot(131)

change\_neuron\_num(50,ax4)

ax5 = plt.subplot(132)

change\_neuron\_num(75,ax5)

ax6 = plt.subplot(133)

change\_neuron\_num(100,ax6)

图表, 散点图

描述已自动生成

图表, 散点图

描述已自动生成

以上结果显示，当网络层数保持不变时，神经元数量的增加并不总是带来更好的效果。增多的神经元数量会使决策边界变得更加复杂。

当改变单隐层中的神经元数量时，决策边界可能会显著变化。较少的神经元可能导致较为简单的决策边界，无法很好地拟合复杂的数据模式。而增加神经元数量则可以提供更大的模型容量，使决策边界能够更好地适应训练数据，从而捕捉数据集中的细节。

然而，需要注意的是，过多的神经元可能会导致过拟合，即模型变得过于复杂，从而无法很好地泛化到新数据。因此，在选择神经元数量时，需要在模型复杂度和泛化能力之间找到平衡。

通过可视化决策边界，可以直观地观察这种影响。增加神经元数量会使决策边界变得更加曲折和详细，而较少的神经元则可能导致边界更加简单和平滑。

【讨论二】 改变神经网络深度（如深度为 2，每层 10 个神经元） ， 其他参数不变， 与讨论一进行对比， 观察神经网络深度其对决策边界的影响。

# 修改隐藏层深度

hidden\_layer\_depths = [1, 2, 3, 4, 5, 6, 7, 8, 9]  # 不同的隐藏层深度

fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(12, 10))

for depth, ax in zip(hidden\_layer\_depths, axes.flatten()):

    # 创建MLP分类器

    clf = MLPClassifier(hidden\_layer\_sizes=(15,) \* depth, max\_iter=1000, random\_state=1111)

    # 拟合模型

    clf.fit(X\_train, y\_train)

    # 绘制决策边界

    x\_min, x\_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5

    y\_min, y\_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

    xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.02),

                         np.arange(y\_min, y\_max, 0.02))

    Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)

    ax.contourf(xx, yy, Z, alpha=0.8)

    # 绘制样本点

    ax.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')

    # ax.set\_title(f"Depth: {depth},train acc={clf.score(X,y)},test acc={clf.score(X\_test,y\_test)}")

    # 像上面这样不好，对于score应该只显示两位小数

    ax.set\_title(f"Depth: {depth},train acc={clf.score(X,y):.3f},test acc={clf.score(X\_test,y\_test):.3f}")

plt.tight\_layout()

plt.show()

图片包含 散点图

描述已自动生成

改变神经网络的深度（增加层数）会对决策边界产生显著影响。与单隐层网络相比，增加网络深度可以提供更高层次的特征表示能力，有助于学习更复杂和抽象的数据模式。

通过增加网络深度，每一层都可以学习数据的不同抽象特征。初始层可以捕捉原始数据的基本特征，而随着网络深度的增加，后续层能够逐步学习更高层次的特征。这样的层次结构有助于构建更复杂的决策边界，使模型更好地区分不同类别的数据点。

然而，增加网络深度也可能带来一些问题。过深的网络可能会遇到梯度消失或梯度爆炸的问题，导致训练困难。此外，过深的网络还会增加模型的复杂度和计算成本。

通过可视化决策边界，可以观察到增加网络深度后，决策边界变得更加复杂，具有更高的分离能力。不同类别的数据点可能会被更细致的边界分隔，使得模型能够更好地适应训练数据。

【讨论三】 在讨论一（或讨论二） 的基础上， 改变激活函数（如 tanh、 relu） ， 与讨论一（或讨论二）进行对比， 观察不同激活函数对决策边界的影响。

# 修改隐藏层深度

hidden\_layer\_depths = [2, 5, 8, 11]  # 不同的隐藏层深度

act\_fun\_list = ['logistic', 'identity', 'relu', 'tanh']

for act\_fun in act\_fun\_list:

    fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(14, 3.4))

    for one\_depth, ax in zip(hidden\_layer\_depths, axes.flatten()):

        # 创建MLP分类器

        clf = MLPClassifier(activation=act\_fun, hidden\_layer\_sizes=(15,) \* one\_depth, max\_iter=1000, random\_state=1)

        # 拟合模型

        clf.fit(X\_train, y\_train)

        # 绘制决策边界

        x\_min, x\_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5

        y\_min, y\_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

        xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.02),

                            np.arange(y\_min, y\_max, 0.02))

        Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])

        Z = Z.reshape(xx.shape)

        ax.contourf(xx, yy, Z, alpha=0.8)

        # 绘制样本点

        ax.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')

        # ax.set\_title(f"Depth: {depth},train acc={clf.score(X,y)},test acc={clf.score(X\_test,y\_test)}")

        # 像上面这样不好，对于score应该只显示两位小数

        ax.set\_title(f"act\_fun:{act\_fun},Depth: {one\_depth}\ntrain acc={clf.score(X,y):.3f},test acc={clf.score(X\_test,y\_test):.3f}")

    plt.tight\_layout()

plt.show()

图表, 散点图

描述已自动生成

图表, 散点图

描述已自动生成

散点图

低可信度描述已自动生成

图表, 散点图

描述已自动生成

改变激活函数会对决策边界产生显著影响。不同的激活函数具有不同的非线性特性，这直接影响神经网络的学习能力和模型的表达能力。

例如，使用tanh激活函数可以产生平滑的非线性特性，取值范围在[-1, 1]之间。这种非线性特性使神经网络能够学习相对复杂且平滑的决策边界。相比于讨论一或讨论二中使用的默认激活函数，tanh激活函数可能会导致更光滑的决策边界。

另一方面，使用ReLU（Rectified Linear Unit）激活函数则会产生更为简单的非线性特性。ReLU在正值范围内保持线性，在负值范围内为零。这种特性使神经网络能够学习到更简单和分段线性的决策边界。相比于讨论一或讨论二中使用的默认激活函数，ReLU激活函数可能会导致更加锐利和角度分明的决策边界。

通过可视化决策边界，可以观察到不同激活函数对边界形状和复杂度的影响。tanh激活函数可能会产生平滑的决策边界，而ReLU激活函数可能会产生更加锐利的边界。选择适当的激活函数取决于数据集的特性、任务的要求以及对决策边界形状的期望。

从relu到tanh到logistic，决策边界越来越光滑，决策边界接触面积越来越小。这与激活函数自身的变化趋势有关。

图表

描述已自动生成 图表

描述已自动生成 图表

描述已自动生成

【讨论四】在讨论三的基础上，增大正则化系数，观察正则化对决策边界的影响。

import numpy as np

import matplotlib.pyplot as plt

from sklearn.neural\_network import MLPClassifier

from sklearn.datasets import make\_moons

from sklearn.model\_selection import train\_test\_split

# 生成示例数据集

X, y = make\_moons(noise=0.3, random\_state=0)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# 不同的隐藏层深度

hidden\_layer\_depths = [2, 5, 8, 11]

# 仅使用relu激活函数

act\_fun = 'relu'

# 增加更多的正则化系数

alphas = [0.0001, 0.001, 0.01, 0.1, 0.5, 1.0]

for alpha in alphas:

    fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(14, 3.4))

    for one\_depth, ax in zip(hidden\_layer\_depths, axes.flatten()):

        # 创建MLP分类器

        clf = MLPClassifier(activation=act\_fun, hidden\_layer\_sizes=(15,) \* one\_depth, alpha=alpha, max\_iter=1000, random\_state=1)

        # 拟合模型

        clf.fit(X\_train, y\_train)

        # 绘制决策边界

        x\_min, x\_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5

        y\_min, y\_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

        xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.02),

                             np.arange(y\_min, y\_max, 0.02))

        Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])

        Z = Z.reshape(xx.shape)

        ax.contourf(xx, yy, Z, alpha=0.8)

        # 绘制样本点

        ax.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')

        ax.set\_title(f"Depth: {one\_depth}\nalpha={alpha}, train acc={clf.score(X\_train, y\_train):.3f}, test acc={clf.score(X\_test, y\_test):.3f}")

    plt.tight\_layout()

    plt.show()

图表, 散点图

描述已自动生成

图表, 散点图

描述已自动生成

图表, 散点图

描述已自动生成

图表, 散点图

描述已自动生成

图表, 散点图

描述已自动生成

图表, 散点图

描述已自动生成

较小的正则化系数（如 0.0001, 0.001）：影响较小，模型更倾向于拟合训练数据，决策边界会更加复杂，可能导致过拟合。

中等的正则化系数（如 0.01, 0.1）：提供适度的约束，有助于模型的泛化，决策边界较为平滑，复杂度适中。

较大的正则化系数（如 0.5, 1.0）：约束较强，限制了模型的复杂度，决策边界变得更简单，可能导致欠拟合。

2、题目二： 采用 scikit-learn 中的 MLPClassifier 对自带手写数字数据集进行分类

from sklearn.neural\_network import MLPClassifier

from sklearn.datasets import load\_digits

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

import numpy as np

from ipywidgets import interactive

from IPython.display import display

from sklearn.model\_selection import validation\_curve,GridSearchCV

import time

X, y = load\_digits(return\_X\_y=True)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=1)

clf = MLPClassifier(learning\_rate\_init=0.0001,random\_state=52, max\_iter=10000).fit(X\_train, y\_train)

proba\_test = clf.predict\_proba(X\_test)

pre\_train = clf.predict(X\_train)

pre\_test = clf.predict(X\_test)

# print("pre:",pre\_test)

# print("gt: ",y\_test)

tp\_train = (pre\_train==y\_train).sum()

tp\_test = (pre\_test==y\_test).sum()

print("train acc: ",tp\_train/len(y\_train))

print("test acc: ",tp\_test/len(y\_test))



【讨论五】 结合模型复杂度与模型泛化误差之间的关系，调节模型超参数，提升模型泛化性能。可尝试调节隐层神经元个数和隐层数、激活函数、学习率、 正则项系数等超参数。

首先调整神经元个数和激活函数类型。

clf = MLPClassifier(activation='logistic',random\_state=2023, max\_iter=1000)

param\_range = np.arange(10,400,10)

start = time.time()

train\_scores, val\_scores = validation\_curve(clf, X\_train, y\_train, param\_name='hidden\_layer\_sizes', scoring='accuracy', \

                                            param\_range=param\_range, cv=5, n\_jobs=-1)

end = time.time()

# Plot the validation curve

train\_mean = np.mean(train\_scores, axis=1)

val\_mean   = np.mean(val\_scores, axis=1)

plt.figure(figsize=(6, 3))

plt.plot( param\_range, train\_mean, color = 'r', label = 'Train')

plt.plot( param\_range, val\_mean,   color = 'g', label = 'Val')

plt.legend(loc='best')

plt.xlabel('神经元个数')

plt.ylabel('正确率')

plt.title('激活函数为logistic')

plt.show()

print("logistic time: ",end-start)

# 绘制出的两条曲线分别是训练集和验证集的准确率随着神经元个数的变化曲线，

# 随着神经元个数的增加，训练集的准确率一直在增加至1，但是验证集的准确率在达到一定值后就不再增加开始波动，出现过拟合现象。

# 由图也可以看到最优值可选为210左右

clf = MLPClassifier(activation='tanh',random\_state=2023, max\_iter=1000)

param\_range = np.arange(10,400,10)

start = time.time()

train\_scores, val\_scores = validation\_curve(clf, X\_train, y\_train, param\_name='hidden\_layer\_sizes', scoring='accuracy', \

                                            param\_range=param\_range, cv=5, n\_jobs=-1)

end = time.time()

# Plot the validation curve

train\_mean = np.mean(train\_scores, axis=1)

val\_mean   = np.mean(val\_scores, axis=1)

plt.figure(figsize=(6, 3))

plt.plot( param\_range, train\_mean, color = 'r', label = 'Train')

plt.plot( param\_range, val\_mean,   color = 'g', label = 'Val')

plt.legend(loc='best')

plt.xlabel('神经元个数')

plt.ylabel('正确率')

plt.title('激活函数为tanh')

plt.show()

print("tanh time: ",end-start)

clf = MLPClassifier(random\_state=2023, max\_iter=1000)   # 这里使用默认激活函数relu

param\_range = np.arange(10,400,10)

start = time.time()

train\_scores, val\_scores = validation\_curve(clf, X\_train, y\_train, param\_name='hidden\_layer\_sizes', scoring='accuracy', \

                                            param\_range=param\_range, cv=5, n\_jobs=-1)

end = time.time()

# Plot the validation curve

train\_mean = np.mean(train\_scores, axis=1)

val\_mean   = np.mean(val\_scores, axis=1)

plt.figure(figsize=(6, 3))

plt.plot( param\_range, train\_mean, color = 'r', label = 'Train')

plt.plot( param\_range, val\_mean,   color = 'g', label = 'Val')

plt.legend(loc='best')

plt.xlabel('神经元个数')

plt.ylabel('正确率')

plt.title('激活函数为relu')

plt.show()

print("relu time: ",end-start)

# 将激活函数更换为tanh（左）、logistic（右）的效果如下，折线图相对稳定一点。此外，选择这两种激活函数时消耗时间约为relu的1.5倍左右

# 综合来看，logistic作激活函数时，结果更加稳定、收敛更快，神经元数量在150时达到验证集准确率上限0.98

图表, 折线图

描述已自动生成

logistic time: 75.30306077003479

图表, 折线图

描述已自动生成

tanh time: 31.96909189224243

relu time: 26.105326652526855 图表, 折线图

描述已自动生成  
综合来看，logistic作激活函数时，结果更加稳定、收敛更快，神经元数量在150时达到验证集准确率上限0.98

接下来调节学习率，使用logistic作激活函数

clf = MLPClassifier(activation='logistic',random\_state=2023, max\_iter=1000)

param\_range = np.arange(0.005,1,0.01)

train\_scores, val\_scores = validation\_curve(clf, X\_train, y\_train, param\_name='learning\_rate\_init', scoring='accuracy', \

                                            param\_range=param\_range, cv=5, n\_jobs=-1)

# Plot the validation curve

train\_mean = np.mean(train\_scores, axis=1)

val\_mean   = np.mean(val\_scores, axis=1)

plt.figure(figsize=(7, 5))

plt.plot(param\_range, train\_mean, color = 'r', label = 'Train')

plt.plot(param\_range, val\_mean,   color = 'g', label = 'Val')

# plt.xticks([0,1,2,3,4],['10e-5','10e-4','10e-3','10e-3','10e-1',])

plt.legend(loc='best')

plt.xlabel('lr\_rate')

plt.ylabel('accuracy')

plt.show()

图表, 折线图

描述已自动生成

这里得到的结果是学习率为0.05时，训练集和验证集的准确率都是很高的，

当学习率过小时，模型的收敛速度会变得非常缓慢，需要更多的迭代次数才能达到较高的准确率。在本例中，由于限定了最大迭代次数为1000次，应该足够让模型收敛到最优解了

随着学习率的增加，准确率一直在下降，这可能是学习率过大，导致模型无法收敛，出现欠拟合现象

当学习率过大时，每次参数更新的幅度会很大，可能导致模型在训练过程中发生不稳定的震荡或发散现象，使得模型无法收敛到最优解，从而降低了准确率

接下来调节隐藏层深度

clf = MLPClassifier(activation='logistic',random\_state=2023, max\_iter=1000)

depth\_range = [1, 2, 3, 4, 5, 6, 7, 8, 9]  # 不同的隐藏层深度

# param\_range\_ = [[int(10)]\*int(i) for i in param\_range]

param\_range = [(15,) \* depth for depth in depth\_range]

train\_scores, val\_scores = validation\_curve(clf, X\_train, y\_train, param\_name='hidden\_layer\_sizes', scoring='accuracy', \

                                            param\_range=param\_range, cv=5, n\_jobs=-1)

# Plot the validation curve

train\_mean = np.mean(train\_scores, axis=1)

val\_mean   = np.mean(val\_scores, axis=1)

val\_50 = val\_mean

plt.figure(figsize=(7, 5))

plt.plot( depth\_range, train\_mean, color = 'r', label = 'Train')

plt.plot( depth\_range, val\_mean,   color = 'g', label = 'Val')

plt.legend(loc='best')

plt.xlabel('hidden\_layers')

plt.ylabel('accuracy')

plt.show()

图表, 折线图

描述已自动生成

plt.plot(val\_50,label='val\_50')

plt.plot(val\_100,label='val\_100')

plt.plot(val\_150,label='val\_150')

plt.legend(loc='best')

plt.show()

图表, 折线图

描述已自动生成

调节正则化系数L2：

clf = MLPClassifier(activation='relu',random\_state=2023, max\_iter=1000)

param\_range = np.logspace(-5,-1,5)

train\_scores, val\_scores = validation\_curve(clf, X\_train, y\_train, param\_name='alpha', scoring='accuracy', \

                                            param\_range=param\_range, cv=5, n\_jobs=-1)

# Plot the validation curve

train\_mean = np.mean(train\_scores, axis=1)

val\_mean   = np.mean(val\_scores, axis=1)

plt.figure(figsize=(7, 5))

plt.plot( train\_mean, color = 'r', label = 'Train')

plt.plot( val\_mean,   color = 'g', label = 'Val')

plt.xticks([0,1,2,3,4],['10e-5','10e-4','10e-3','10e-3','10e-1',])

plt.legend(loc='best')

plt.xlabel('L2')

plt.ylabel('accuracy')

plt.show()

图表, 折线图

描述已自动生成

clf = MLPClassifier(hidden\_layer\_sizes=220,alpha=0.0001,random\_state=1, max\_iter=10000).fit(X\_train, y\_train)

proba\_test = clf.predict\_proba(X\_test)

pre\_train = clf.predict(X\_train)

pre\_test = clf.predict(X\_test)

# print("pre:",pre\_test)

# print("gt: ",y\_test)

tp\_train = (pre\_train==y\_train).sum()

tp\_test = (pre\_test==y\_test).sum()

print("train acc: ",tp\_train/len(y\_train))

print("test acc: ",tp\_test/len(y\_test))



确定出较为具体的参数范围：神经元个数：100~200，网络层数：1，激活函数：logistic，学习率：0.001（默认），正则项系数：默认

3、 题目三： 编写 BPNN 算法， 对 iris 数据集/手写数字集进行二分类或多分类。

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_digits

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import confusion\_matrix

class BPNN(object):

    def \_\_init\_\_(self,n\_features,n\_kinds,neus\_each\_layer,activation='relu'):

        # 根据样本特征数、隐藏层数、每层神经元数量，随机初始化权重和偏置

        self.w1 = np.random.randn(neus\_each\_layer,n\_features)

        self.b1 = np.random.randn(neus\_each\_layer,1)

        self.w2 = np.random.randn(neus\_each\_layer,neus\_each\_layer)

        self.b2 = np.random.randn(neus\_each\_layer,1)

        self.w3 = np.random.randn(n\_kinds,neus\_each\_layer)

        self.b3 = np.random.randn(n\_kinds,1)

        self.act = self.relu if activation=='relu' else self.sigmoid

        self.act\_diff = self.relu\_diff if activation=='relu' else self.sigmoid\_diff

        # print("init:")

        # print("w1",self.w1.tolist())

        # print("b1",self.b1.tolist())

        # print("w2",self.w2.tolist())

        # print("b2",self.b2.tolist())

    def feedforward(self, x,  ):                       #前向传播

        out1 = np.dot(self.w1, x) + self.b1

        out1 = self.act(out1)     # 返回每层的激活值

        out2 = np.dot(self.w2, out1) + self.b2

        out2 = self.act(out2)

        out3 = np.dot(self.w3, out2) + self.b3

        out3 = self.act(out3)

        return out1, out2, out3

    def backpropagation(self,X,gt,out1,out2,out3):            #反向传播

        # e = out3-gt

        e = (1-gt)/(1-out3)-gt/out3

        f1 = self.act\_diff(out1)

        f2 = self.act\_diff(out2)

        f3 = self.act\_diff(out3)

        # print(e)

        # print(f1)

        # print(f2)

        # print("e",e.shape)

        # print("f1",f1.shape)

        # print("f2",f2.shape)

        # print("out1",out1.shape)

        # print("out2",out2.shape)

        # 计算输出层神经元梯度

        grad\_w3 = np.dot(e\*f3,out2.T)/X.shape[1]

        grad\_b3 = (e\*f3).mean(axis=1).reshape(-1,1)

        # 计算隐层神经元梯度

        temp = e\*f3

        temp = np.dot(self.w3.T,temp)\*f2

        grad\_w2 = np.dot(temp,out1.T)/X.shape[1]

        grad\_b2 = temp.mean(axis=1).reshape(-1,1)

        temp = np.dot(self.w2.T,temp)\*f1

        grad\_w1 = np.dot(temp,X.T)/X.shape[1]

        grad\_b1 = temp.mean(axis=1).reshape(-1,1)

        return grad\_w1,grad\_b1,grad\_w2,grad\_b2,grad\_w3,grad\_b3        #返回每层的权重梯度和偏置梯度

    def update\_weight(self,grad\_w1,grad\_b1,grad\_w2,grad\_b2,grad\_w3,grad\_b3,learning\_rate=0.01):   #更新参数

        self.w1 -= grad\_w1 \* learning\_rate

        self.b1 -= grad\_b1 \* learning\_rate

        self.w2 -= grad\_w2 \* learning\_rate

        self.b2 -= grad\_b2 \* learning\_rate

        self.w3 -= grad\_w3 \* learning\_rate

        self.b3 -= grad\_b3 \* learning\_rate

    def loss(self,pre,gt): # 代价函数

        return - (1-gt)\*np.log(1-pre) - gt\*np.log(pre)

    def relu(self, x):

        return np.maximum(x, 0)

    def relu\_diff(self, x):    #relu函数导数 输入为 f(wx+b)

        return np.where(x>0,1,0)

    def sigmoid(self, x):

        return 1/(1+np.exp(-x))

    def sigmoid\_diff(self, x):   #sigmoid函数导数 输入为 f(wx+b)

        # sig = self.sigmoid(x)

        # return sig\*(1-sig)

        return x\*(1-x)

    def fit(self, X, y,num\_epoches=100, learning\_rate=0.01):   #训练过程

        loss\_list = []

        # num\_epoches为迭代次数

        for epoch\_id in range(num\_epoches):

            lr = learning\_rate\*(1-float(epoch\_id)/num\_epoches)\*\*1.5 #动态学习率

            if epoch\_id%100==0:

                print('\*'\*20,'iter',epoch\_id,'\*'\*20)

            a = np.random.choice(y.shape[1],[20,int(y.shape[1]/20)],replace=False)

            # 1.经典的四步训练流程：前向计算->计算损失->计算梯度->更新参数（分别调用类的方法）

            for index in a:

                out1,out2,out3 = self.feedforward(X[:,index])  # out为激活之后的

                grad\_w1,grad\_b1,grad\_w2,grad\_b2,grad\_w3,grad\_b3 = self.backpropagation(X[:,index],y[:,index],out1,out2,out3)

                self.update\_weight(grad\_w1,grad\_b1,grad\_w2,grad\_b2,grad\_w3,grad\_b3,learning\_rate=lr)

            # 2.将每次迭代的代价函数的值存起来，用于绘制损失曲线（也可在循环中输出每次迭代的损失值，查看中间过程）

                loss = self.loss(out3,y[:,index])

                loss\_list.append(loss.sum())

            # 3.可将损失函数数组/列表返回

        return loss\_list

    def predict(self,X):

        out1,out2,out3 = self.feedforward(X)

        # print(np.round(out3,1).tolist()[0])

        # print(np.round(out3,1).tolist()[1])

        # print(np.round(out3,1).tolist()[2])

        return np.argmax(out3,axis=0)

X,y = load\_digits(return\_X\_y=True)

y = np.eye(np.unique(y).size)[y]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=6)

clf = BPNN(X.shape[1],y.shape[1],neus\_each\_layer=40,activation="rel")

loss = clf.fit(X\_train.T,y\_train.T,num\_epoches=800,learning\_rate=0.6)

pre\_train = clf.predict(X\_train.T)

print('\ntest:')

pre\_test = clf.predict(X\_test.T)

gt\_train = y\_train.argmax(axis=1)

gt\_test = y\_test.argmax(axis=1)

tp\_train = (pre\_train==gt\_train).sum()

tp\_test = (pre\_test==gt\_test).sum()

print("train acc: {:>4}/{:>4}={:.3f}".format(tp\_train, gt\_train.size, tp\_train/gt\_train.size))

print("test  acc: {:>4}/{:>4}={:.3f}".format(tp\_test, gt\_test.size, tp\_test/gt\_test.size))

plt.plot(loss)

plt.xlabel('epoch')

plt.ylabel('loss')

plt.show()

文本

描述已自动生成

形状, 正方形

描述已自动生成

关键代码片段：

前向传播：

第一层：

activation

        out1 = np.dot(self.w1, x) + self.b1

        out1 = self.act(out1)     # 返回每层的激活值

第二层：

activation

out2 = np.dot(self.w2, out1) + self.b2

        out2 = self.act(out2)

输出层：

activation

out3 = np.dot(self.w3, out2) + self.b3

        out3 = self.act(out3)

梯度计算：

对输出层权重和偏置的梯度：

grad\_w3 = np.dot(e\*f3,out2.T)/X.shape[1]

        grad\_b3 = (e\*f3).mean(axis=1).reshape(-1,1)

对隐藏层权重和偏置的梯度：

temp = e\*f3

        temp = np.dot(self.w3.T,temp)\*f2

        grad\_w2 = np.dot(temp,out1.T)/X.shape[1]

        grad\_b2 = temp.mean(axis=1).reshape(-1,1)

对输入层权重和偏置的梯度：

temp = np.dot(self.w2.T,temp)\*f1

        grad\_w1 = np.dot(temp,X.T)/X.shape[1]

        grad\_b1 = temp.mean(axis=1).reshape(-1,1)

参数更新：

使用梯度下降法更新权重和偏置：

self.w1 -= grad\_w1 \* learning\_rate

        self.b1 -= grad\_b1 \* learning\_rate

        self.w2 -= grad\_w2 \* learning\_rate

        self.b2 -= grad\_b2 \* learning\_rate

        self.w3 -= grad\_w3 \* learning\_rate

        self.b3 -= grad\_b3 \* learning\_rate

激活函数：

交叉熵损失函数：

def loss(self,pre,gt): # 代价函数

        return - (1-gt)\*np.log(1-pre) - gt\*np.log(pre)

Relu函数：

图片包含 图示

描述已自动生成

def relu(self, x):

        return np.maximum(x, 0)

    def relu\_diff(self, x):    #relu函数导数 输入为 f(wx+b)

        return np.where(x>0,1,0)

Sigmod函数：

def sigmoid(self, x):

        return 1/(1+np.exp(-x))

    def sigmoid\_diff(self, x):   #sigmoid函数导数 输入为 f(wx+b)

        # sig = self.sigmoid(x)

        # return sig\*(1-sig)

        return x\*(1-x)

手动计算：

图示, 文本

描述已自动生成