PJ1（反向传播算法）说明文档

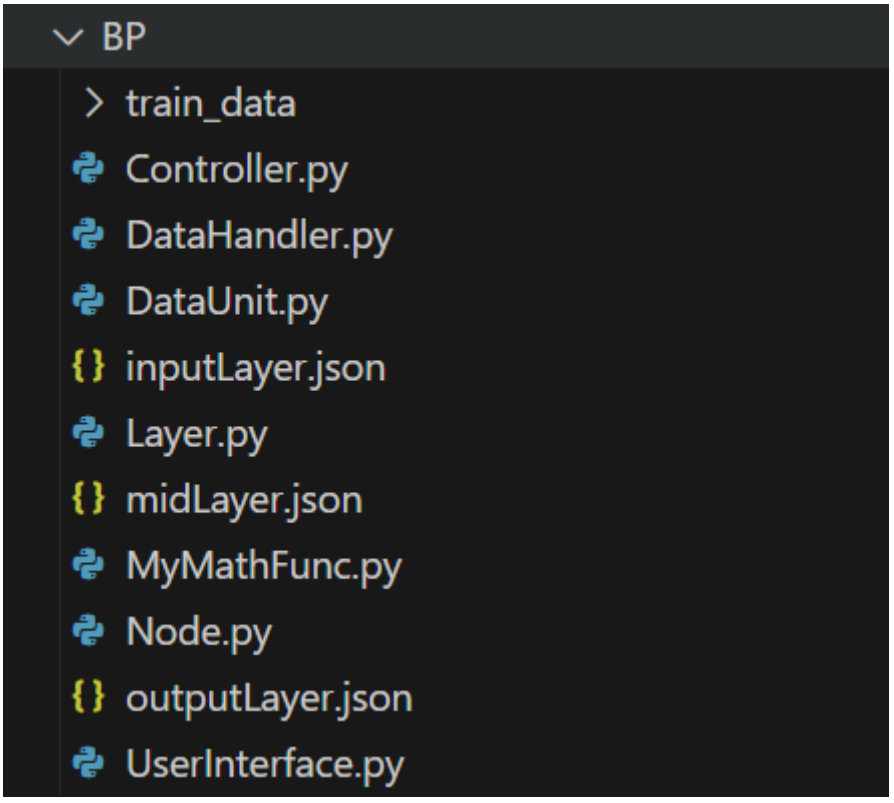
22302010019 陈星宇

一、整体结构

1.文件结构

BP神经网络进行sin函数拟合、BP神经网络进行12个手写体汉字识别、CNN进行12个手写体汉字识别的相关代码分别在Code/Sin、Code/BP、Code/CNN文件夹下。以下仅说明BP神经网络的文件结构。

12个手写体汉字识别任务中，有以下文件：



说明如下：

train_data：数据集，从中划分出训练集和测试集

UserInterface.py：控制台交互接口，用于读写训练数据时获取用户选择

MyMathFunc.py：数学工具方法的集合，执行部分数学运算，如error计算

inputLayer.json、midLayer.json、outputLayer.json：保存的训练数据，含各层各个结点偏置与权重

DataUnit.py：数据元类，有数据pixels和类型category两个成员变量

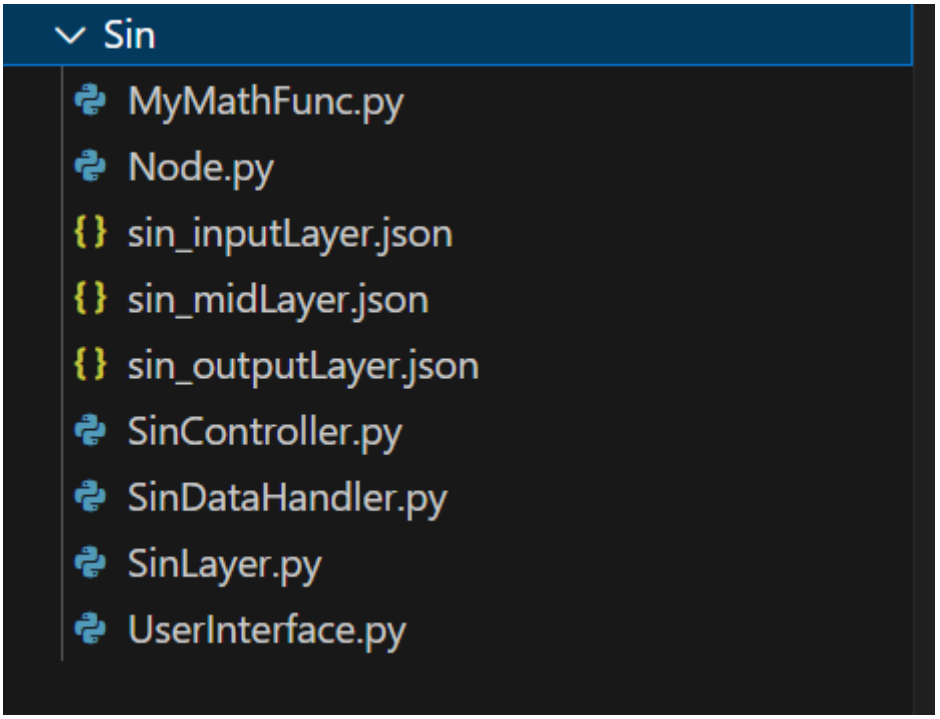
DataHandler.py：数据处理方法的集合，用于读入数据和数据预处理、存写训练后的数据

Node.py：神经元结点类，内部用字典存储了一系列参数，如权重、偏置、梯度、输入输出等

Layer.py：神经层类，内部包含若干结点，含有前向传播、反向传播等BP神经网络的核心方法

Controller.py：主控类，用于组织网络结构，控制训练与测试过程

sin函数拟合的任务较之于12个手写体汉字识别的任务更加简单，网络结构和具体算法上做了一定简化。各个文件作用同上,不再赘述。sin函数拟合的任务的样本取自[-π，π]区间上sin曲线的随机点，共1000个。测试集为[-π，π]区间上的1000个值，通过神经网络输出预测的1000个对应y值。



2.网络结构

两个神经网络都有输入层、中间层、输出层三层。BP神经网络进行12个手写体汉字识别任务中，三层分别有[28*28,32,12]个结点；BP神经网络进行sin函数拟合任务中，三层分别有[1,7,1]个结点。其中中间层结点的选择来源于经验公式和实际调试：　　a+√（m+n）（m、n为输入输出层结点数，a为0~9的常数）

节点数的确定由主控Controller的超参数指定，各层网络存储于Controller中的layers变量内，学习率由layer中的超参数指定，实现了可伸缩易调整的网络结构。

二、代码运行方法

先选择是否使用训练好的参数，再确定是否用仅测试的模式运行，如下图：

```
PS D:\大二下基础课与专业课\人工智能\Lab\Lab1\Code> python -u "d:\大二下基础课与专业课\人工智能\Lab\Lab1\Code\BP\Controller.py"
数据处理完成
训练集已加载完成。train_size: 6600  total batch: 550
测试集已加载完成。test_size: 840
是否从文件中加载神经网络？请输入Y/y或N/n: y
是否仅测试现有训练数据？请输入Y/y或N/n: wadhjoaihds(错误输入示例)
输入错误，请重新输入。
是否仅测试现有训练数据？请输入Y/y或N/n:
```

程序结束前，会询问是否将当前参数写入文件：

```
7  out:    0.074598718695577 66
8  out:    0.064587979687641 27
9  out:    0.080716720566922 32
10  out:    0.071302780208838 36
11  out:    0.070565858906200 4
correct!category:  3 Num:  839
Testing process done. CorrectRate: 91.54762%
是否把现有的训练成果写入文件？请输入Y/y或N/n: y
现有的训练成果已写入JSON文件
```

三、关键方法说明：

(此处的代码均为12个手写体汉字识别中使用的反向传播算法，sin函数拟合中使用的算法为其简化版本以加快训练速度)

1.反向传播过程：

```
 #反向传播方法，计算输入、中间层神经元的delta,中间、输出层的delta_bias
def backward(layers,expectOutput):
        inputLayer=layers[0]
        midLayer=inputLayer.rightLayer
```

```
        outputLayer=midLayer.rightLayer

        for i in range(0,outputLayer.nodeNum):# 遍历输出神经元
            outputNode=outputLayer.neurons[i]

outputNode.params['delta_bias']+=b_learningRate*outputNode.old_tanh
_derivative()*(expectOutput[i]-outputNode.params['output'])

        for j in range(0, midLayer.nodeNum):# 遍历中间层神经元
            midNode=midLayer.neurons[j]
            outParam=0
            for i in range(0,outputLayer.nodeNum):# 遍历输出神经元

                outputNode=outputLayer.neurons[i]
                outParam+=midNode.params['weight']
[i]*outputNode.old_tanh_derivative()*(expectOutput[i]-
outputNode.params['output'])
                midNode.params['delta_weight']
[i]+=w_learningRate*outputNode.old_tanh_derivative()*
(expectOutput[i]-outputNode.params['output'])
*midNode.params['output']

 midNode.params['temp_delta_bias']=b_learningRate*outParam*
                    midNode.tanh_derivative()

 midNode.params['delta_bias']+=midNode.params['temp_delta_bias']

        for k in range(0, inputLayer.nodeNum):# 遍历输入层神经元
            inputNode=inputLayer.neurons[k]
            for j in range(0,midLayer.nodeNum):# 遍历中间神经元
                midNode=midLayer.neurons[j]
                inputNode.params['delta_weight']
[j]+=midNode.params['temp_delta_bias']*inputNode.params['output']
```

其中，outParam对应的是以下算式的值：

$$\sum_i w_{ji}(O_i(1-O_i))\cdot(d_i-O_i)$$

中间层结点的delta_bias参数和输入层结点的delta_weight参数之间仅相差一个输入层结点的输出值，故考虑使用一个中间变量midNode.params['temp_delta_bias']存储这个结点本次反向传播的delta_bias，可供计算inputNode.params['delta_weight']时复用。实验证明，这种复用使整体训练速度提升了30%以上。

2.输出层输出调整：

选取sigmoid作为激活函数时：（现已弃用下面的方法）

```
# 对于输出层，调整比例，使各个神经元输出和为1（旧方法，现已弃用）
def getRealOutput(self)->None:
    if(self.rightLayer is not None):
        print("错误调用getRealOutput：不是输出层")
        return None
    outputSum=0
    for i in range(0,self.nodeNum):
        outputSum+=self.neurons[i].params['output']
    for i in range(0,self.nodeNum):

self.neurons[i].params['output']=self.neurons[i].params['output']/o
utputSum
```

选取tanh作为激活函数时（输出层原始输出区间为[-1,1]，上述方法不再适用，改用softmax方法）：其中，params['old_output']存储的是其未经softmax的输出值，这是为了保证tanh_derivative的值不会受softmax的影响，在上面的方法中也忽略了这一点，导致计算的梯度实际上是不准确的，这点在实验中也得到了验证：

本方法未引入params['old_output']时，取12*550个样本构成样本集，其余为测试集，学习率0.03，在15个epoch后，测试集上的平均正确率为44.05%，loss在0.44左右，在100个epoch后，测试集上的平均正确率为88.01%，loss在0.39左右。在保留了未经softmax的输出值计算梯度后，以同样参数训练，15个epoch后，测试集上的平均正确率已经高达为78.81%，loss在0.397左右。可以看出，这个忽略对实验结果的影响是巨大的。

```
def softmax(self)->None:
        if(self.rightLayer is not None):
            print("错误调用getRealOutput：不是输出层")
            return None
        outputSum=0
        for i in range(0,self.nodeNum):
            outputSum+=np.exp(self.neurons[i].params['output'])
        for i in range(0,self.nodeNum):

 self.neurons[i].params['old_output']=self.neurons[i].params['outpu
t']

 self.neurons[i].params['output']=np.exp(self.neurons[i].params['ou
tput'])/outputSum
```

四、激活函数、超参数的探索与选择

1.激活函数

一开始，两个实验中都选取sigmoid作为激活函数，但是，对于sin函数的拟合中，发现由于sigmoid函数产生的激活值值域不关于0对称，导致后层的神经元的输入是非0均值的信号，这会对梯度产生影响，从而导致拟合效果差。改为tanh作为激活函数，并改变对应导数后，拟合效果有很大提升。

2.超参数

（1）权重

两个实验中，权重初始化都为正态初始化：

```
def initWeight(weightNum:int):
      normalizedWeight=np.random.normal(loc=0, scale=0.1,
size=weightNum)
      return np.clip(normalizedWeight,a_min=-1/35,a_max=1/35)
```

即，结点权重在[-1/35,1/35]间，且其散布服从均值为0，方差0.1的正态分布，这种初始化权重的方法可以帮助改善模型的数值稳定性、降低过拟合风险，并且有助于更稳定地进行训练。

在12个手写体汉字识别任务中，单纯地取[-1/35,1/35]区间内的随机数作为结点权重，在其他参数不变的情况下，训练5个epoch，正确率为35.05%；加入正则化，训练5个epoch，正确率为39.41%。当然，这一差距是可以通过增加训练次数弥补的，但也证明这种初始化权重的方法有一定优势。

（2）偏置

由经验，设置中间层偏置为[-0.2,0.2]间的随机数，输出偏置为[-0.1,0]间的随机数。

```
mid_min_bias=-0.2
mid_max_bias=0.2
output_min_bias=-0.1
output_max_bias=0
```

（3）学习率

12个手写体汉字识别任务中，取不同的学习率，其他参数控制相同，分别训练15个epoch，测试集上的正确率变化如下：

LEARNING_RATE	CORRECT_RATE
1	82.03%
0.1	86.79%
0.05	86.49%
0.03	84.08%
0.01	77.02%

为防止梯度消失/爆炸，最终选择使用0.05作为初始学习率，error下降速度有明显放缓时，适当降低学习率。

（4）批大小

合理选择batch_size可以提高训练效率、稳定性和泛化性能，经调试定为24（sin函数拟合中最终定为20）

（5）中间层节点数

12个手写体汉字识别任务中，取不同的中间层结点数，其他参数控制相同，分别训练15个epoch，测试集上的正确率变化如下：

MIDLAYER节点数	CORRECT_RATE
20	77.19%
25	82.92%
30	83.01%
34	85.03%
35	81.09%
37	81.44%
1100	44.03%(可能发生了过拟合)

最终选定midLayer节点数为34，这也符合经验公式a+√（m+n）。

五、实验结果：

12个手写体汉字识别：

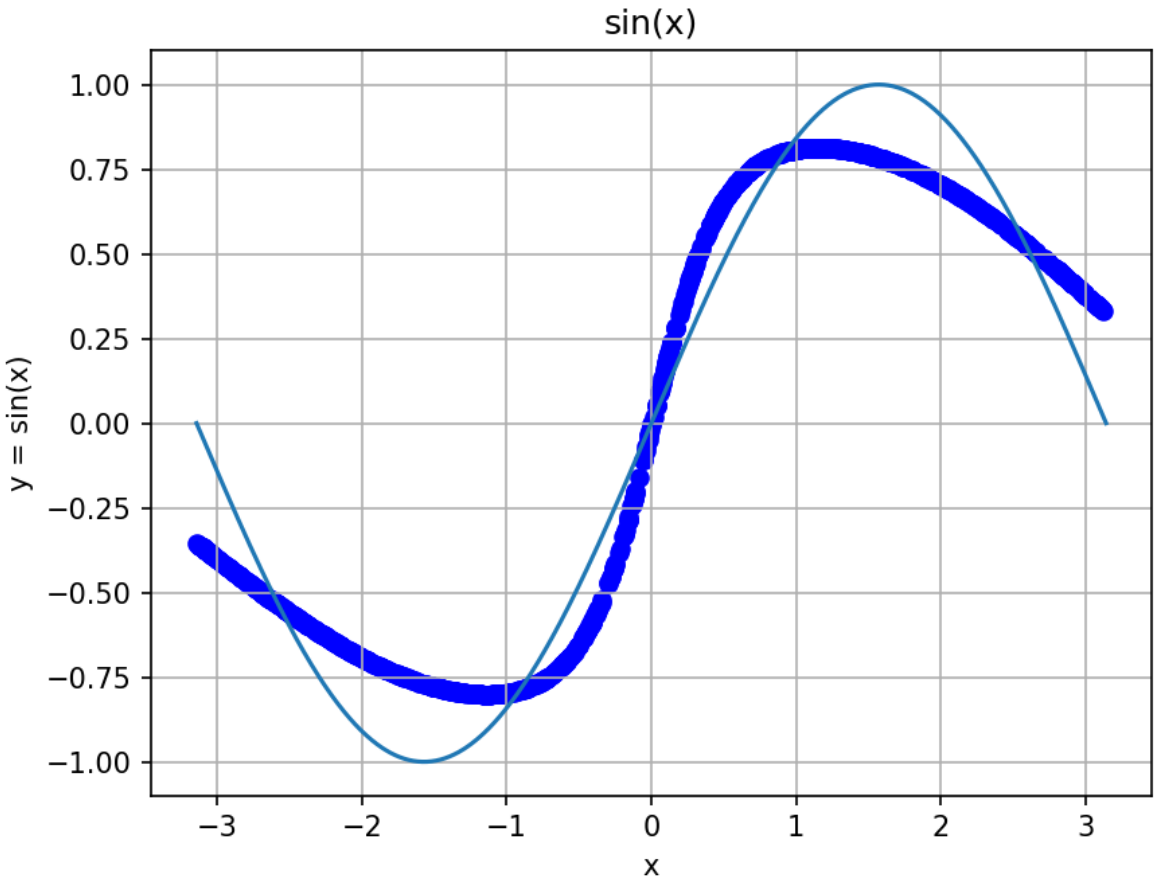
使用上述探究最终所得的超参数组合，训练过程如下：

EPOCH	AVERAGE ERROR	CORRECT RATE	NOTE
15	0.397	78.71%	学习率0.05，batch_size24
100	0.381	82.24%	同上
300	0.374	88.03%	同上
350	0.368	94.03%	同上
650	0.367	95.15%	同上

继续训练，进步已经不太显著。

sin函数拟合：

拟合（-π，π）区间上的sin函数，取1000个样本点，学习率0.05，在1000个epoch后，平均error为0.145左右（此处的error计算方法为math.fabs(expectOutput-realOutput)）。图像如下：

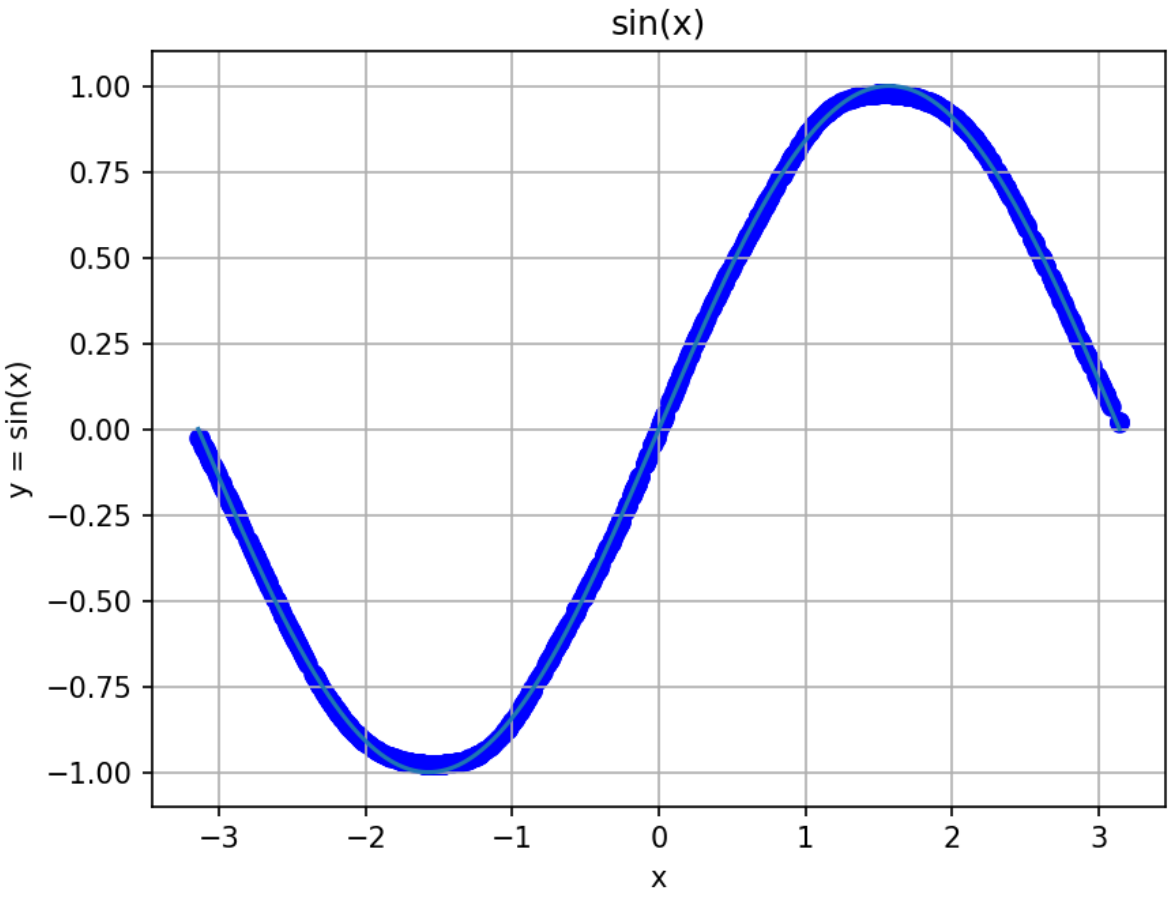


训练过程如下：

EPOCH	AVERAGE ERROR	NOTE
1000	0.1452	学习率0.005，batch_size为1000（一个epoch仅更新1次）
100000	0.0161	同上
220000	0.0130	同上
330000	0.0125	同上
340000	0.0120	学习率0.001，batch_size为20（一个epoch更新50次）
350000	0.0115	学习率0.002，batch_size为20（一个epoch更新50次）
450000	0.0083	同上

（Average loss=0.0083时已达本实验精度要求，故不再继续迭代）

Average loss=0.0083时，拟合图像如下：



六、其他说明

1.对于数据集，以下几个样本与真实的字形有严重偏离：1.554，3.555，4.45，5.604，处理方法为删除这些字，用与真实字形相近的其他样本代替。

2.运行环境：

操作系统：Windows 11

Python版本：3.12.2

IDE：VSCode 1.86

依赖项：numpy 1.26.4

matplotlib 3.8.4

七、反思与总结

本次实验耗时相当之久，让我对反向传播算法有了深入的理解：反向传播算法通过前向传播和反向传播两个过程，实现了神经网络模型的训练和参数优化。与CNN对比，反向传播算法对输入有更高的适应度，但是由于缺少卷积、池化的过程，泛化能力明显更弱。哪怕问题规模很小，也需要大量样本和反复训练。从实现细节上看，设置合适的学习率是反向传播算法中的关键，过大的学习率可能导致震荡或发散，而过小的学习率可能导致收敛速度过慢；此外，选择合适的激活函数能够增强模型的非线性表达能力，也是不可忽视的细节。

本PJ还有以下几点可能可以做出一定改进：

1.将反向传播算法修改为矩阵运算，这不仅使网络的可伸缩性更强，同时可以利用并行计算和矢量化指令集，加快计算速度，此外，将多个参数的更新合并为一次操作，减少了代码的复杂度和冗余性。这样可以更容易理解和维护代码。

2.结点的设计上，有一些空间被浪费，如old_output对非输出层结点并无意义，temp_delta_bias对非中间层结点无意义等。可以考虑用其他的数据结构单独存储以节省空间的开销。

3.可以尝试使用一些方法对学习率进行动态改变，如SGD、Adam等：（下图是SGD）

Bounded stochastic gradients: $\mathcal{O}(1/\varepsilon^2)$ steps

Theorem

Let $f : \mathbb{R}^d \to \mathbb{R}$ be convex and differentiable, $\mathbf{x}^\star$ a global minimum; furthermore, suppose that $\|\mathbf{x}_0 - \mathbf{x}^\star\| \leq R$, and that $\mathbb{E}\left[\|\mathbf{g}_t\|^2\right] \leq B^2$ for all t. Choosing the constant stepsize

$$\gamma := \frac{R}{B\sqrt{T}}$$

stochastic gradient descent yields

$$\frac{1}{T}\sum_{t=0}^{T-1} \mathbb{E}\big[f(\mathbf{x}_t)\big] - f(\mathbf{x}^\star) \leq \frac{RB}{\sqrt{T}}.$$

4.可以尝试使用可视化方法展示error的下降过程，有助于理解和调试超参数。