智能系统原理与开发 Lab1 文档

第一部分：测试使用说明

第一步：运行根目录下的 fountain.py 文件可以完成第一部分有关 sinx 函数的拟合
第二步：将图片测试集放于文件根目录
第三部：打开 main.py 文件，并将其中的目录改为测试文件夹目录

```python
from test import display_test
from transform import convert_img_to_csv

IMG_DIR = r"C:\Users\LYZ\Documents\Code\BP_algorithm\test"  #在此处修改为测试图片所在的地址

convert_img_to_csv(IMG_DIR)  #将会在项目根目录生成test.csv文件

display_test()
```

第四步：运行 main.py 文件

第二部分：代码思路解释

一、sinx 函数的拟合

对于拟合函数来说其 loss 函数的形式如下：

$$E(y,\overline{y}) = \frac{\sum_{i=1}^{n}(y-\overline{y})^2}{n}$$

BP 网络结构的定义，可以在__init__里实现：

```python
def __init__(self, hidden_size=100):
    self.params = {'W1': np.random.random((1, hidden_size)),
                   'B1': np.zeros(hidden_size),
                   'W2': np.random.random((hidden_size, 1)),
                   'B2': np.zeros(1)}
```

这里我们把数据组织形状为 (samples, features)这样的形式，samples 代表数据点个数，features 代表每个数据点的特征数，在这里因为只有一个自变量，特征数为 1，如果我们产生 100 个数据点，那么数据的形状是(100,1)，我们可以一次行把这个矩阵数据输入神经网络，即上面的输入层神经元个数为 100. 为了产生这样的训练数据，可以利用文章 np.random 用法里面的 Numpy 方法，在类定义里增加产生数据的函数 generate_data，可以为不同函数产生训练数据集。

```python
def generate_data(fun, is_noise=True, axis=np.array([-1, 1, 100])):
    np.random.seed(0)
    x = np.linspace(axis[0], axis[1], axis[2])[:, np.newaxis]
    x_size = x.size
    y = np.zeros((x_size, 1))
    if is_noise:
        noise = np.random.normal(0, 0.1, x_size)
    else:
        noise = None

    for i in range(x_size):
        if is_noise:
            y[i] = fun(x[i]) + noise[i]
        else:
            y[i] = fun(x[i])

    return x, y
```

激活函数选择 sigmoid

```python
@staticmethod
def sigmoid(x_):
    return 1 / (1 + np.exp(-x_))

def sigmoid_grad(self, x_):
    return (1.0 - self.sigmoid(x_)) * self.sigmoid(x_)
```

从输入到输出结果计算，隐藏层用激活函数 sigmoid，输出层不用激活函数，实现 predict 方法如下：

```python
def predict(self, x_):
    W1, W2 = self.params['W1'], self.params['W2']
    b1, b2 = self.params['B1'], self.params['B2']

    a1 = np.dot(x_, W1) + b1
    z1 = self.sigmoid(a1)
    a2 = np.dot(z1, W2) + b2

    return a2
```

因为是对比输出值的偏差，所以选择均方差 MSE 作为损失函数或者叫优化函数，定义 loss 函数定义实现

```python
def loss(self, x_, t):
    y_ = self.predict(x_)
    return y_, np.mean((t - y_) ** 2)
```

神经网络是根据计算出来的 loss 不断调整权重和偏置，以便不断减少 loss 达到逼近函数的目的。

```
def gradient(self, x, t):
    W1, W2 = self.params['W1'], self.params['W2']
    b1, b2 = self.params['B1'], self.params['B2']
    grads = {}

    batch_num = x.shape[0]

    # forward
    a1 = np.dot(x, W1) + b1
    z1 = self.sigmoid(a1)
    a2 = np.dot(z1, W2) + b2

    # backward
    dy = (a2 - t) / batch_num
    grads['W2'] = np.dot(z1.T, dy)
    grads['B2'] = np.sum(dy, axis=0)

    dz1 = np.dot(dy, W2.T)
    da1 = self.sigmoid_grad(a1) * dz1
    grads['W1'] = np.dot(x.T, da1)
    grads['B1'] = np.sum(da1, axis=0)
    return grads
```

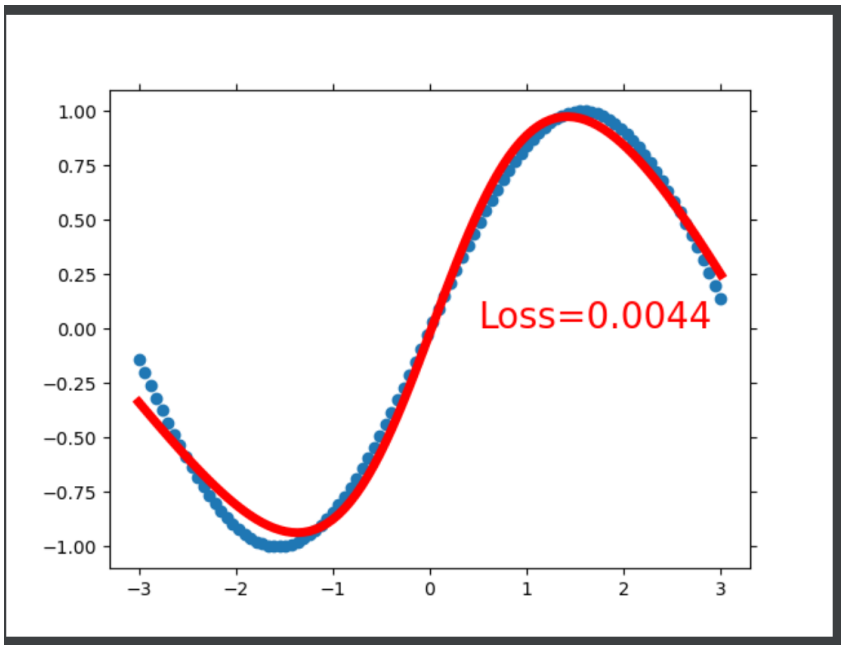
有了上面的步骤，就可以定义训练主函数，并且将训练的结果展示出来

```
def train_with_own(self, x_, y_, max_steps=100):
    for k in range(max_steps):
        grad = self.gradient(x_, y_)
        for key in ('W1', 'B1', 'W2', 'B2'):
            self.params[key] -= self.r * grad[key]

        pred, loss = self.loss(x_, y_)
        if k % 150 == 0:
            plt.cla()
            plt.scatter(x_, y_)
            plt.plot(x_, pred, 'r-', lw=5)
            plt.text(0.5, 0, 'Loss=%.4f' % abs(loss), fontdict={'size': 20, 'color': 'red'})
            plt.pause(0.1)

    plt.ioff()
    plt.show()
```

实验结果如下：



二、手写汉字分类

为了实现可自定义层数、节点数、学习率等基于 BP 的易于伸缩的网络模型，我们需要对第一部分的代码进行很大程度的修改，故一二部分的核心代码并不相同。
BPNeuralNetwork.py 文件为该网络的核心实现部分。
为了可复用定义多层结构，所以大部分参数的定义都要延后实现，代码如下：

```
class NeuralNetwork:  # 神经网络
    def __init__(self, layers):  # layers为神经元个数列表
        self.activation = sigmoid  # 激活函数
        self.activation_deriv = sigmoid_derivative  # 激活函数导数
        self.weights = []  # 权重列表
        self.bias = []  # 偏置列表
        for i in range(1, len(layers)):  # 参数初始化
            self.weights.append(np.random.randn(layers[i - 1], layers[i]))
            self.bias.append(np.random.randn(layers[i]))
```

有关激活函数的定义：

```
def sigmoid(x):  # 激活函数采用Sigmoid
    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(x):  # Sigmoid的导数
    return sigmoid(x) * (1 - sigmoid(x))
```

有关学习率衰减，选择的为根据 epochs 进行指数衰减，有关各层的正反向传播都改为了根据 lay 进行修改，而不是直接写死，有关训练的主函数如下：

```
def fit(self, x, y, learning_rate=0.2, epochs=3):  # 反向传播算法
    x = np.atleast_2d(x)
    n = len(y)  # 样本数
    p = max(n, epochs)  # 样本过少时根据epochs减半学习率
    y = np.array(y)

    for k in trange(epochs * n):  # 带进度条的训练过程
        if (k + 1) % p == 0:
            learning_rate *= 0.5  # 每训练完一代样本减半学习率
        a = [x[k % n]]  # 保存各层激活值的列表
        # 正向传播开始
        for lay in range(len(self.weights)):
            a.append(self.activation(np.dot(a[lay], self.weights[lay]) + self.bias[lay]))
        # 反向传播开始
        label = np.zeros(a[-1].shape)
        label[y[k % n]] = 1  # 根据类号生成标签
        error = label - a[-1]  # 误差值
        deltas = [error * self.activation_deriv(a[-1])]  # 保存各层误差值的列表

        layer_num = len(a) - 2  # 导数第二层开始
        for j in range(layer_num, 0, -1):
            deltas.append(deltas[-1].dot(self.weights[j].T) * self.activation_deriv(a[j]))  # 误差的反向传播
        deltas.reverse()
        for i in range(len(self.weights)):  # 正向更新权值
            layer = np.atleast_2d(a[i])
            delta = np.atleast_2d(deltas[i])
            self.weights[i] += learning_rate * layer.T.dot(delta)
            self.bias[i] += learning_rate * deltas[i]
```

互联网上的训练集数据很多都是以.scv 文件形式存在，并且在实现过长中发现直接读取图片的速度相对读取处理好的 scv 文件要快一些，所以此处的输入选择 scv 文件。

```
def train():
    file_name = 'train.csv'
    y = []
    x = []
    y_t = []
    x_t = []
    with open(file_name, 'r') as f:
        reader = csv.reader(f)
        header_row = next(reader)
        print(header_row)
        for row in reader:
            if np.random.random() < 0.8:  # 80%的数据用于训练
                y.append(int(row[0]))
                x.append(list(map(int, row[1:])))
            else:
                y_t.append(int(row[0]))
                x_t.append(list(map(int, row[1:])))
    len_train = len(y)
    len_test = len(y_t)
    print('训练集大小%d，测试集大小%d' % (len_train, len_test))
    x = np.array(x)
    y = np.array(y)
    nn = NeuralNetwork([784, 200, 12])  # 神经网络各层神经元个数
    nn.fit(x, y, 0.05, 200)
    file = open('NN.txt', 'wb')
    pickle.dump(nn, file)
    count = 0
    for i in range(len_test):
        p, _ = nn.predict(x_t[i])
        if p == y_t[i]:
            count += 1
    print('模型识别正确率：', count / len_test)
```

可以看到在下图所示位置我们可以轻松定义所需的网络结构并自定义参数，举例如下：

```
nn = NeuralNetwork([784, 500, 200, 12])  # 神经网络各层神经元个数
nn.fit(x, y, 0.05, 200)
```

上图所示为一个四层结构的 BP 网络，第一隐层有 500 个节点，第二隐层有 300 个节点。学习率为 0.2，迭代轮数为 200。

在讲图片文件转换为 scv 文件的过程中，将灰度反转，避免大量灰度为 255 的像素点出现，可有效避免大数的遮蔽效应，代码如下：

```
def convert_img_to_csv(img_dir):
    with open(r'test.csv', 'w',
              newline='') as f:
        column_name = ['label']
        column_name.extend('pixel%d' % i for i in range(64 * 64))

        writer = csv.writer(f)
        writer.writerow(column_name)

        for i in range(1, 13):
            img_file_path = os.path.join(img_dir, str(i))
            img_list = os.listdir(img_file_path)

            for img_name in img_list:
                img_path = os.path.join(img_file_path, img_name)
                img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
                img = img - 255
                img *= 255
                image_data = [i -1]
                image_data.extend(img.flatten())
                # print(image_data)
                writer.writerow(image_data)
```

第三部分：实现过程中遇到的问题

在测试中发现，对于简单手写体识别这一任务来说，增加层数和节点数对提高正确率都有所帮助，但是提升都很有限，而学习率和迭代次数与初始 bios 则是决定正确率的关键。

譬如，若不采用学习率指数下降的办法，则会严重拉低正确率；增加迭代次数会有助于提高正确率；在此样例中，bios 为负值，并且；在参数的初始化中，使用随机正则生成的参数往往能达到更高的正确率。