第一部分：反向传播算法

计算机科学技术学院　　周训哲 20307110315

2023 年 3 月 18 日

一、算法原理

反向传播算法的原理是利用反向传播的误差进行梯度下降，然后不断更新权重，直至各个神经节点所存储的权重以及偏置能够根据输入值进行正向传播较好拟合真实输出。

二、代码架构

主体：反向传播算法

具体而言，反向传播算法主要有初始化、正向传播、反向传播、预测、计算误差五个部分组成。

初始化

首先在初始化过程中，神经网络会根据输入值的维度、输出值的维度、中间隐藏层的层数、维度生成相应的权重矩阵、偏置矩阵，其中存储的值可以在一开始设定为符合标准正态分布的随机数。然后设置对应的超参数以及激活函数。

```
def __init__(self, input_dim, hidden_layer, output_dim, batch_size,
        batches, learning_rate, task_type, l2_lambda, dropout):
```

```
        if task_type not in ('regression', 'classification'):
                raise ValueError('Now only \'regression\' model \
                        and \'classification\' model are supported!')
        self.input_dim = input_dim
        self.hidden_layer = hidden_layer
        self.output_dim = output_dim
        self.batch_size = batch_size
        self.batches = batches
        self.learning_rate = learning_rate
        self.shrink_lr = False
        self.l2_lambda = l2_lambda
        self.dropout = dropout
        self.activation_function = relu
        if task_type == 'regression':
                self.classification_function = self.activation_function
        elif task_type == 'classification':
                self.classification_function = softmax
        self.loss_function = loss
        self.w = {}
        self.b = {}
        self.layers = len(hidden_layer)
        self.w[0] = np.random.randn(input_dim, hidden_layer[0])
        for i in range(1, self.layers):
                self.w[i] = np.random.randn(hidden_layer[i-1], hidden_layer[i])
        self.w[self.layers] = np.random.randn(hidden_layer[self.layers-1],
                 output_dim)
        for i in range(self.layers):
                self.b[i] = np.zeros((1, hidden_layer[i]))
        self.b[self.layers] = np.zeros((1, output_dim))
```

正向传播

正向传播过程中，输入层通过网络正向传播，每经过一层隐藏层节点，上一层数据都会利用权重矩阵和偏置矩阵进行线性计算，然后将得到的值再使用激活函数进行下一步计算，得到该层的输出并传输到下一层，直至最后的输出。

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \tag{1}$$
$$\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)}) \tag{2}$$
$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} \tag{3}$$
$$\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)}) \tag{4}$$
$$\vdots \tag{5}$$
$$\mathbf{z}^{(L)} = \mathbf{W}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)} \tag{6}$$
$$\hat{\mathbf{y}} = \sigma(\mathbf{z}^{(L)}) \tag{7}$$

```
def forward(self, x):
        self.z = {}
        self.a = {}
        self.z[0] = np.dot(x, self.w[0]) + self.b[0]
        self.a[0] = self.activation_function.forward(self, self.z[0])
        for i in range(1, self.layers):
                self.z[i] = np.dot(self.a[i-1], self.w[i]) + self.b[i]
                self.a[i] = self.activation_function.forward(self, self.z[i])
        self.z[self.layers] = np.dot(self.a[self.layers-1],
                 self.w[self.layers]) + self.b[self.layers]
        self.a[self.layers] = self.classification_function.forward(self,
                 self.z[self.layers])
        return self.a[self.layers]
```

反向传播

反向传播则是正向传播的逆过程。首先计算输出与真实值之间的误差作为反向传播的输入值。然后利用误差的链式法则乘上激活函数的导数实现激活函数的反向求导。计算出最后的误差后，利用误差可以计算权重值以及偏置的微分，再用误差继续反向传播，直至算出最后一层隐藏层的误差值。然后再用微分对权重和偏置进行更新以实现梯度下降。

首先计算输出层的误差：

$$\delta^{(L)} = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y}) \odot \sigma'(\mathbf{z}^{(L)}) \tag{8}$$

然后，通过误差逐层反向传播计算损失的梯度：

$$\delta_i^{(l)} = \begin{cases} \delta_i^{(L)} & \text{if } l = L \\ \left(\sum_{j=1}^{n_{l+1}} w_{i,j}^{(l)} \delta_j^{(l+1)} \right) \odot \sigma'(\mathbf{z}_i^{(l)}) & \text{otherwise} \end{cases} \tag{9}$$

根据损失的梯度可以计算权重与偏置的梯度：

$$\nabla_{w_{i,j}^{(l)}} L = \delta_j^{(l+1)} a_i^{(l)} \tag{10}$$

$$\nabla_{b_i^{(l)}} L = \delta_i^{(l)} \tag{11}$$

最后利用梯度更新权重与偏置：

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \alpha \nabla_{\mathbf{W}^{(l)}} L \tag{12}$$

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \alpha \nabla_{\mathbf{b}^{(l)}} L \tag{13}$$

```
def backward(self, x, y, y_hat):
        delta = {}
        dw = {}
        db = {}
        delta[0] = loss.mean_square(self, y, y_hat) *
                self.classification_function.backward(self, y_hat)
```

```
        dw[0] = np.dot(self.a[self.layers - 1].T, delta[0])
        db[0] = np.sum(delta[0], axis=0, keepdims=True)
        for i in range(self.layers-1, 0, -1):
                delta[self.layers-i] = np.dot(delta[self.layers-i-1],
                        self.w[i+1].T) *\
                        self.activation_function.backward(self, self.a[i])
                dw[self.layers-i] = np.dot(self.a[i-1].T, delta[self.layers-i])
                db[self.layers-i] = np.sum(delta[self.layers-i],
                        axis=0, keepdims=True)
        delta[self.layers] = np.dot(delta[self.layers-1], self.w[1].T) *\
         self.activation_function.backward(self, self.a[0])
        dw[self.layers] = np.dot(x.T, delta[self.layers])
        db[self.layers] = np.sum(delta[self.layers], axis=0)
        # # 缩小学习率
        # if (dw[0].max() < 1.0 or dw[0].min() > -1.0) and
        #         not self.shrink_lr:
        #     self.shrink_lr = True
        #     self.learning_rate /= 10
        # 正则化
        for i in range(self.layers+1):
                dw[i] += self.l2_lambda * self.w[self.layers - i]
                self.w[i] -= self.learning_rate * dw[self.layers-i]
                self.b[i] -= self.learning_rate * db[self.layers-i]
```

预测与计算误差

预测过程则是利用最后的权重与偏置对输入最后进行一次正向传播得到神经网络预测的输出。计算误差则是选定一个合适的损失函数计算预测值与真实输出之间的误差。

```
def predict(self, x):
        y_hat = self.forward(x)
        return y_hat
```

```
# sin
loss = np.square(np.subtract(y_hat, y)).mean()
```

```
# hanzi
error_count = 0.0
for i in range(len(y_hat)):
        index = np.argmax(y_hat[i])
        if y_test[i][index] != 1:
                error_count += + 1.0
accuracy = 1 - (error_count / len(y_hat))
```

代码整体架构：

在本次实验中，我实现了模型的复用，通过设置可选参数实现灵活调整网络结构，诸如网络节点层的结构可以通过输入对应的列表实现，输入、输出层的维度可以通过参数实现，批处理的大小以及学习的循环次数可以通过参数设定，一些超参数如学习率、正则化系数、dropout 百分比、最后一层的激活函数都可以通过参数确定。

除开模型的架构以外，主函数的内容就是设定输入、输出集，对网络进行训练，对输出误差进行计算和评判，输出相关统计图像。

```
# sin
if __name__ == '__main__':
np.random.seed(0)
```

```
bp = BP(input_dim=1, hidden_layer=[100, 100], output_dim=1,
        batch_size=batch_size, batches=batches, learning_rate=0.00001,
        task_type='regression', l2_lambda=0.01, dropout=0.7)
bp.train('sin', x_train.reshape(-1, 1), y_train.reshape(-1, 1), epochs,
        x_test.reshape(-1, 1), y_test.reshape(-1, 1))
y_hat = bp.predict(x_test.reshape(-1, 1))
y_hat = y_hat.reshape(1, y_hat.shape[0])[0]
loss = np.square(np.subtract(y_hat, y_test)).mean()
print('mean square error: ', loss)

plt.scatter(x_test, y_test, label='True')
plt.scatter(x_test, y_hat, label='Predicted')
plt.text(0.5, -0.3, 'MSE error = %f ' % loss, size=10, ha='center')
plt.legend()
plt.show()
```

```
# hanzi
if __name__ == '__main__':
        np.random.seed(0)
        bp = BP(input_dim=784, hidden_layer=[500, 300, 300, 100], output_dim=12,
                batch_size=batch_size, batches=batches, learning_rate=0.001,
                task_type='classification', l2_lambda=0.01, dropout=0.7)
        bp.train('hanzi', x_train, y_train, epochs, x_test, y_test)
        y_hat = bp.predict(x_test)
        error_count = 0.0
        for i in range(len(y_hat)):
                index = np.argmax(y_hat[i])
                if y_test[i][index] != 1:
                        error_count += + 1.0
        accuracy = 1 - (error_count / len(y_hat))
        print('accuracy: ', accuracy)
```

三、参数、方法介绍

本模型主要使用的参数包括学习率、正则化系数、批处理大小、dropout 百分比、神经网络节点数与层数、训练次数、激活函数类型。

	sin 函数	手写汉字识别
学习率	0.001	0.001
正则化系数	0	0.01
批处理大小	80	40
dropout 百分比	0.7	0.7
训练次数	10000	500
神经网络节点数与层数	[100, 100]	[500, 300, 300, 100]
激活函数类型	每一层使用的都是 tanh	除了最后一层使用的 softmax，其余均使用的 tanh

说明

sin 函数拟合中经过交叉验证寻找均方误差最低的时候的训练次数，发现随着训练次数的增加，均方误差越小，所以网络训练时选取一个相对较大但是又不耗时的 10000 次。

手写汉字识别经过交叉验证收敛次数大概为 8000，但是峰值大概在 500 左右，并且 500 次训练出来的结果准确率也较高，所以网络采用的训练时间更少的 500 次

四、网络结构、参数实验过程

sin 函数拟合

在实验过程中，首先在 100 个单层节点训练 10000 次的条件下使用了 sigmoid 激活函数，但是该激活函数对于负的值处理效果非常不好，会导致

负值经过激活函数的值全变成 0，所以最后的输出就是原点左边为 0，右边输出为正常的拟合值，最后的均方误差大概是 0.20。

然后使用了 leaky relu 激活函数，结果较之前有所好转，但是对于负数部分的预测仍然出现问题，导致最后预测的结果在原点左边是一条斜率大于 0 的直线，均方误差最后为 0.1。最后尝试了 tanh 作为激活函数，预测的均方误差一下子就降低到 0.001，效果极佳。

仔细研究，如果使用 sigmoid 或 relu 函数作为最后一层的激活函数，会使最后一层的输出中对于负数的值分布在激活函数的左半部分，导致误差上升。所以后续首先尝试在最后一层到输出层之间不添加任何激活函数，首先对于 sigmoid 来说，均方误差一下子就下降到 0.000052；对于 relu 而言，则出现了梯度爆炸的现象，降低学习率之后，均方误差也能达到 0.0016；对于 tanh 来说，也出现了梯度爆炸的现象，调整学习率后，均方误差变成了 0.003。

由以上测试可以看到，sigmoid 函数的拟合能力更强，之后只针对 sigmoid 函数，增加网络层数使用两层 100 个节点，再次进行测试，均方误差只有 0.000017，继续增加层数变成 3 层，均方误差为 0.000075，较两层有增加，说明可能是出现了过拟合的现象。

最后进行交叉验证两层节点的均方误差寻找最优的训练次数，发现当训练次数越多，均方误差越小，所以，选取 100000 次训练观察结果居然可以达到 0.000002。

网络结构	[100]	[100]	[100]
激活函数	sigmoid	relu	tanh
学习率	0.001	0.001	0.001
训练次数	10000	10000	10000
批处理大小	80	80	80
其他处理方法	批处理 + 随机梯度下降	批处理 + 随机梯度下降	批处理 + 随机梯度下降
均方误差	0.2	0.1	0.001

[100]	[100]	[100]
sigmoid	relu	tanh
0.001	0.00001	0.00001
10000	10000	10000
80	80	80
批处理 + 随机梯度下降	批处理 + 随机梯度下降	批处理 + 随机梯度下降
最后一层不使用激活函数	最后一层不使用激活函数	最后一层不使用激活函数
0.000052	0.0016	0.003

[100, 100]	[100, 100, 100]	[100]
sigmoid	sigmoid	tanh
0.001	0.001	0.00001
10000	10000	100000
80	80	80
批处理 + 随机梯度下降	批处理 + 随机梯度下降	批处理 + 随机梯度下降
最后一层不使用激活函数	最后一层不使用激活函数	最后一层不使用激活函数
0.000017	0.000075	0.000002

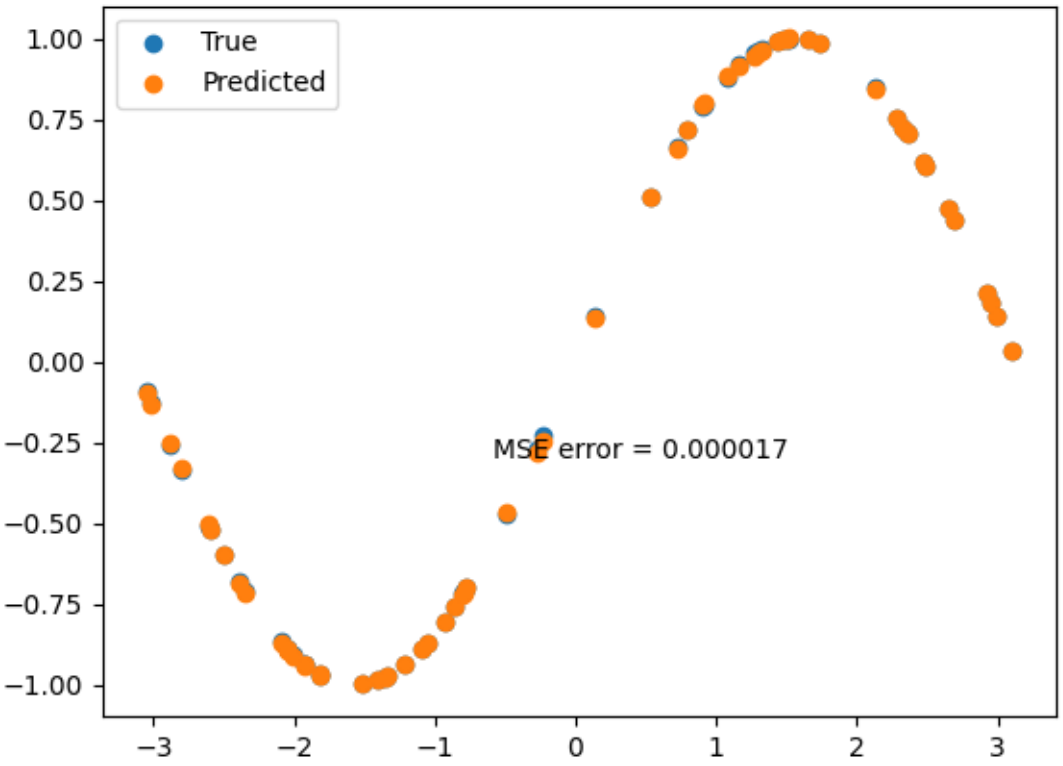

图 1: 使用 sigmoid 激活函数拟合的 sin 函数随机点与原函数对比

手写汉字识别

对于手写汉字识别任务来说，首先使用的是单层的模型，并且对数据没有任何的处理，只是简单的随机分为训练集和测试集，并且一开始为了测试网络的拟合能力，使用的是 10% 的数据作为训练集，90% 的数据作为测试集。

对于 sigmoid 和 relu 函数来说，由于他们在原点左边的几乎没有梯度。同时 relu 函数在原点右边的梯度增长十分迅速，所以很快 relu 激活函数就出现了梯度爆炸的现象。

在第一次训练中，各激活函数的预测准确率基本上都是在 10% 左右，即相当于没有预测的准确率。发现原因可能是没有随机化训练集，或者网络结构的优化梯度出现问题，或者权重设置出现问题。

后续在训练过程中加入了批处理以及随机梯度下降的方法，relu 的准确率为 10% 左右，基本为没有预测，sigmoid 的准确率在 37% 左右。于是尝试使用 tanh 作为中间节点的激活函数，能够基本达到 46% 左右的准确率。

后续测试过程中，就基本敲定使用 tanh 作为网络的激活函数。尝试将节点层数增加到两层，能够将准确率提升到 56% 左右。同时再使用三层网络发现准确率大概在 65% 左右，四层网络最优，66%。五层网络及以后的层数增加只会降低准确率，原因可能是由于增加节点层数导致过拟合的现象。

之后将网络的输出层的激活函数进行替换，更改为 softmax 函数后，发现准确率没有明显提升，但是输出变得更加规范，即输出的概率之间的差值更大，更明显。

在实验的时候增加了正则项，发现准确率又往上提升了一点，提升到 72% 左右。

最后将训练集与测试集的比例改成 9：1，发现准确率基本稳定在 80% 以上，当训练次数为 500 左右时达到峰值，但是未收敛，会出现浮动的准确率。当训练次数达到 8000 左右时，准确率会收敛到 83% 左右，并缓慢下降，原因是训练次数增加可能出现过拟合的现象。

网络结构	[100]	[100]	[100]	[100]
激活函数	sigmoid	relu	tanh	sigmoid
学习率	0.001	0.001	0.001	0.001
训练次数	10000	10000	10000	10000
批处理大小	80	80	80	80
其他处理方法	none	none	none	批处理 + 随机梯度下降
准确率	0.083	梯度爆炸	0.083	0.37

[100]	[100]	[100, 100]
leaky relu	tanh	tanh
0.00001	0.001	0.001
10000	10000	10000
80	80	80
批处理 + 随机梯度下降	批处理 + 随机梯度下降	批处理 + 随机梯度下降
0.1	0.46	0.56

[100, 100]	[100, 100, 100]	[100, 100, 100, 100]
tanh+softmax 做分类激活函数	tanh	tanh+softmax
0.001	0.001	0.001
10000	10000	10000
80	80	80
批处理 + 随机梯度下降	批处理 + 随机梯度下降	批处理 + 随机梯度下降
0.56	0.65	0.66

[100, 100, 100, 100]	[100, 100, 100, 100]	[100, 100, 100]
tanh+softmax	tanh+softmax	tanh+softmax
0.001	0.001	0.001
10000	10000	500
80	80	80
批处理 + 随机梯度下降 + 正则化	批处理 + 随机梯度下降 + 正则化 + 训练: 测试 =9:1	批处理 + 随机梯度下降 + 正则化 + 训练: 测试 =9:1
0.72	0.83	0.87（最高）

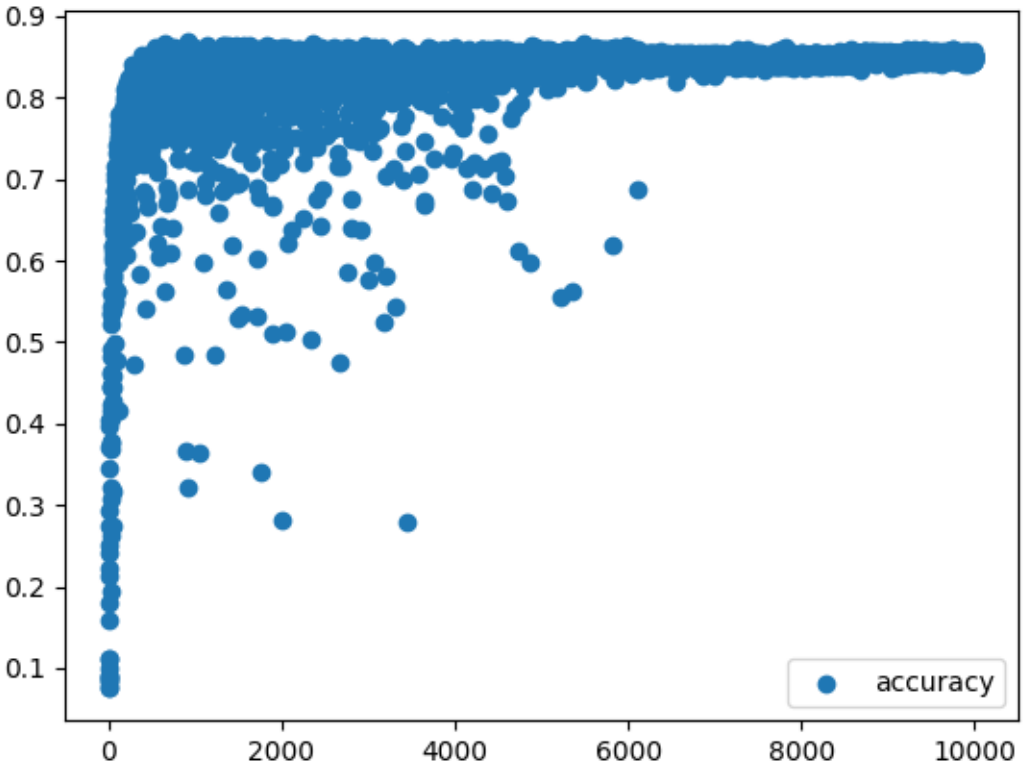

图 2: 准确率随训练次数变化的散点图

五、提高准确率的其他尝试

使用交叉熵损失函数，使用动态的学习率，使用 dropout 方法

六、对反向传播算法的理解

反向传播算法的原理其实就是利用导数以及导数的链式法则对权重以及偏置在向输出误差更小的方向进行梯度优化。但是在优化过程中可能会由于节点层数过少、优化次数过少出现欠拟合的现象。同时也可能由于节点数过多，优化次数过多出现过拟合的现象。除此而外，还会由于激活函数的选择影响输出的结果、出现梯度爆炸的现象等。

反向传播算法是神经网络中最基本的算法，反向传播算法构建的网络也是神经网络中最基本的网络之一。研究反向传播算法可以理解神经网络传播中的一些常见的问题以及处理方法，还可以学习到一些神经网络处理过程中的一些技巧，比如预训练、随机梯度下降、dropout 方法、正则化等。