第 1 章　实验名称:神经网络的反向传播优化算法

1.1　实验目的

1. 理解神经网络(例:简单的前馈神经网络)训练优化算法(反向传播)的数学原理。

2. 理解神经网络反向传播算法的实际运行流程并利用 numpy 库手动实现神经网络反向传播算法。

3. 理解神经网络训练、测试的基本流程。

4. 理解激活函数的作用,验证不同网络结构(线性网络、非线性网络)在处理不同数据分布时的效果差异性。

1.2　实验原理

1.2.1　整体介绍

利用反向传播算法通过链式法则计算损失函数对神经网络中各层参数的梯度,并利用梯度下降法更新网络参数以实现训练神经网络。下图中我们举一个四层(输入层,隐藏层 1,隐藏层 2,输出层)的简单前馈神经网络作为示例(本次实验中,我们即将要实现一个四层神经网络):

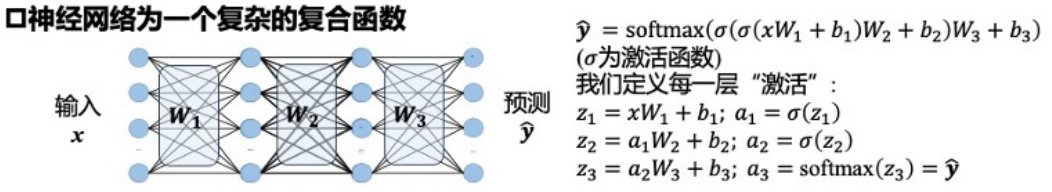

图 1:一个四层前馈神经网络

1.2.2 任务数据介绍

我们想要训练一个四分类的四层前馈神经网络。我们的训练数据包含输入：x（一个 10 维向量：e.g.,$x = (0.872, 0.641, \cdots, -0972)^T$）和输出：一个四分类标签 y（4 维 one-hot 向量，分类对应维度为 1，其他维度对应为 0，e.g.,$y = (0, 1, 0, 0)^T$，即表明该数据应该被分为第二类）。

1.2.3 前向传播过程

我们将 x 输入神经网络，经过前向传播计算后得到神经网络的预测结果 $\hat{y}$:

$$\hat{y} = \text{softmax}(\sigma(\sigma(xW_1 + b_1)W_2 + b_2)W_3 + b_3),$$

其中 $W_{1,2,3}$ 代表神经网络层与层之间链接的权重矩阵；$b_{1,2,3}$ 代表神经网络层与层之间链接权重矩阵的偏置项;$\sigma(\cdot)$ 代表神经网络中使用的激活函数（例如 ReLU 函数）；softmax$(\cdot)$ 代表将神经网络的输出值归一化为和为 1 的概率分布值（如下图所示，输入 softmax 函数之前，网络的输出值，一个 5 维向量，各个维度的值相加之和不为 1，输入之后，网络的输出值被转化为各个维度值大于零且加和为 1 的概率分布向量）。获得预测的概率分布 $\hat{y}$ 后，我们

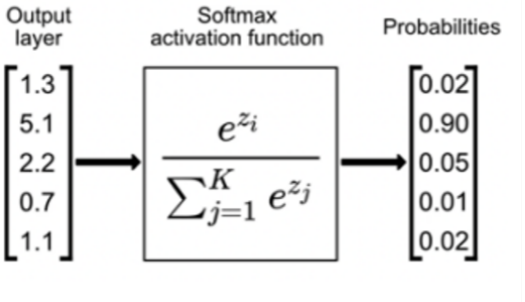

图 2：softmax 函数实例

将其和分类标签 y 结合在一起计算分类误差：我们使用交叉熵（cross entropy）损失函数计算最终的分类误差 $L(y, \hat{y}) = -y \log \hat{y}$.（在本实验中，给出的代码已经含括了交叉熵的计算，求导过程。如同学们想进一步了解交叉熵的计算和求导，可以参考资料：[1]CSDN 博客；[2]博客园博客）。我们用一张图归纳神经网络前向传播过程 (图 3)：

1.2.4 反向传播过程

我们的目标是从后向前逐层计算损失函数 $L(y, \hat{y})$ 对神经网络中的参数 (即：$W_{1,2,3}$ 和 $b_{1,2,3}$) 的梯度。

(1) 计算 W_3,b_3 的梯度：

$$e_3 = \frac{\partial L(y, \hat{y})}{\partial z_3} = \frac{\partial(-y \log \text{softmax}(z_3))}{\partial z_3} = \hat{y} - y;$$

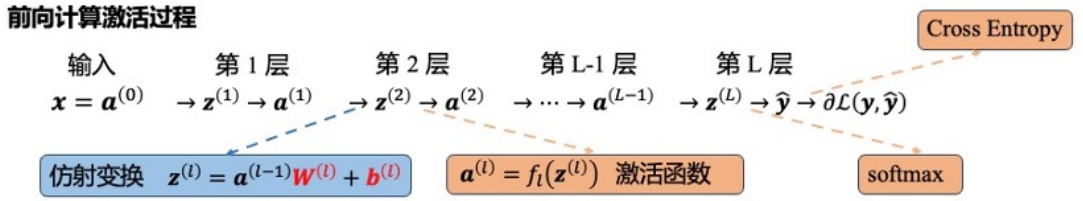

图 3:神经网络前向传播过程示例

$$\frac{\partial L(y,\hat{y})}{\partial W_3} = a_2^T e_3; \frac{\partial L(y,\hat{y})}{\partial b_3} = e_3;$$

(2) 计算 W_2,b_2 的梯度:

$$e_2 = \frac{\partial L(y,\hat{y})}{\partial z_2} = (e_3 W_3^T) \odot \nabla\sigma(z_2);$$
$$\frac{\partial L(y,\hat{y})}{\partial W_2} = a_1^T e_2; \frac{\partial L(y,\hat{y})}{\partial b_2} = e_2;$$

(3) 计算 W_1,b_1 的梯度:

$$e_1 = \frac{\partial L(y,\hat{y})}{\partial z_1} = (e_2 W_2^T) \odot \nabla\sigma(z_1);$$
$$\frac{\partial L(y,\hat{y})}{\partial W_1} = a_0^T e_1 = x^T e_1; \frac{\partial L(y,\hat{y})}{\partial b_1} = e_1;$$

1.2.5　梯度下降法更新网络参数

为了优化神经网络中的参数:$W_{1,2,3}$ 和 $b_{1,2,3}$ 以减小分类损失,我们采用随机梯度下降 (SGD) 方法更新参数:(η 为学习率,$\eta > 0$)

$$W_i = W_i - \eta \frac{\partial L(y,\hat{y})}{\partial W_i},$$
$$b_i = b_i - \eta \frac{\partial L(y,\hat{y})}{\partial b_i}.$$

在实际应用中,我们往往选择在一批 (batch) 数据上进行批量随机梯度下降 (BSGD),假设批为 $\mathcal{B} = \{(x_j, y_j)\}_{j=1}^n$ (n 代表一批中数据的个数),我们的批量随机梯度下降公式为:(η 为学习率,$\eta > 0$)

$$W_i = W_i - \eta(\frac{1}{n}\sum_{j=1}^{n} \frac{\partial L(y_j,\hat{y}_j)}{\partial W_i}),$$
$$b_i = b_i - \eta(\frac{1}{n}\sum_{j=1}^{n} \frac{\partial L(y_j,\hat{y}_j)}{\partial b_i}).$$

1.3 实验工具

在本地计算机完成的环境

(1) python 3.8

(2) numpy 1.24.4

(服务器已装好相关环境（conda 环境：neural_networks），以上只是参考环境，本实验并不对环境有十分严格要求，绝大多数较新版本的 python 和 numpy 都能完成本实验。)

1.3.1 Python 的科学计算库: numpy

NumPy（Numerical Python）是 Python 的一种开源的数值计算扩展。这种工具可用来存储和处理大型矩阵，比 Python 自身的嵌套列表（nested list structure) 结构要高效的多（该结构也可以用来表示矩阵（matrix））, 支持大量的维度数组与矩阵运算，此外也针对数组运算提供大量的数学函数库。本实验主要涉及 numpy 的基础操作和多维数组（来表示神经网络中的输入、输出、激活和参数矩阵等）的相关操作, 请阅读以下 numpy 介绍教程，并尝试运行一些简单的 numpy 函数：

(1) NumPy 常用函数及基础用

(2) Numpy 多维数组的介绍及其常用属性和操作方法

(3) 多维数组矩阵操作介绍

1.4 数据集准备和说明

本实验已经封装好代码进行随机数据生成来进行训练和测试，固不需要利用任何外部数据。具体来说，本实验将会利用线性可分数据和非线性可分数据探究神经网络中非线性激活函数的作用。我们在代码中已经设置好两个函数 generate_linear_data() 和 generate_nonlinear_data()，用来生成线性可分数据和非线性可分数据。

1.5 实验步骤

在本次实验提供的文件 experiment_bp.py (下载地址, 密码：2025ai_bp) 中，我们目标使用 numpy 库手动实现神经网络的反向传播优化训练的代码。学生需要先行阅读基本代码框架理解其基本逻辑（包含数据生成、网络定义、神经网络的训练和测试等几部分），然后根据课堂中所学知识和本实验指导书基本原理部分补全神经网络中前向传播 (forward) 函数和反向传播 (backward) 函数中的关键代码，最后运行补全后的代码并观察模型训练、测试过

程。

1.5.1　环境搭建

1. 在统一服务器上进行实验:执行 ‘conda activate neural_networks‘ 命令即可。

2. 在自己电脑上进行实验:

安装 python 3.8 环境

pip install numpy==1.24

1.5.2　实验实现

```
import numpy as np
import sys

exp_type = sys.argv[1] # 'linear' or 'nonlinear'
if exp_type not in ['linear', 'nonlinear']:
    raise Exception("`exp_type` parameter should be 'linear' or 'nonlinear'.")

np.random.seed(0)
```

首先引入相关的包（numpy, 作为我们实现大多数矩阵定义和计算的工具；sys，作为我们利用命令行参数的工具包）。我们的实验分为两种类型（即在线性可分数据上进行训练和在线性不可分数据上进行训练，两种类型实验均需进行），通过命令行传入“linear”或者“nonlinear”参数来区分。

1. 运行 linear 实验:python experiment_bp.py linear

2. 运行 nonlinear 实验:python experiment_bp.py nonlinear

```
def generate_linear_data(n_samples_per_class=100, n_features=10)

def generate_nonlinear_data(n_samples_per_class=100, n_features=10)

def relu(x)

def relu_derivative(x)

def softmax(x)
```

本文件中已经实现了 5 个函数,供大家调用,其中 generate_linear_data 和 generate_nonlinear_data 分别实现生成线性可分数据和线性不可分数据,以供大家进行 linear 实验和 nonlinear 实验。relu 函数实现了 ReLU 非线性激活函数的计算,relu_derivate 函数实现了 ReLU 函数的求导和 softmax 函数实现了 Softmax 函数的计算。

```
class ReluNeuralNetwork

class LinearNeural Network
```

在文件中，我们定义了两类神经网络，一类（ReluNeuralNetwork）使用 ReLU 函数作为非线性激活函数（因此能够更好地捕捉非线性可分数据的特征）、另一类（LinearNeuralNetwork）则不使用非线性激活函数（因此应该只适用于捕捉线性可分数据的特征）。本次实验中，我们的主要任务即为补全这两个神经网络的前向传播 (forward) 和反向传播 (backward) 代码。

```
# 生成数据供训练、测试使用
if exp_type == 'nonlinear':
    # 生成非线性可分的数据（训练集，测试集）
    X_train, y_train = generate_nonlinear_data(n_samples_per_class=100, n_features=10)
    X_test, y_test = generate_nonlinear_data(n_samples_per_class=100, n_features=10)

else:
    # 生成线性可分的数据（训练集，测试集）
    X_train, y_train = generate_linear_data(n_samples_per_class=100, n_features=10)
    X_test, y_test = generate_linear_data(n_samples_per_class=100, n_features=10)

'''
生成数据信息:
X : 数据矩阵 shape = (n_samples, n_features) = (400, 10)
y : 标签 shape = (n_samples,) = (400,)
'''
print(X_train.shape, y_train.shape)
print('具体例子 X_train[0]:', X_train[0])
print('具体例子 y_train[0]:', y_train[0])

'''
将标签y转换one-hot向量
'''
y_train_onehot = np.eye(4)[y_train]
y_test_onehot = np.eye(4)[y_test]

print('具体例子 y_train_onehot[0]:', y_train_onehot[0])
```

这一部分代码主要用来生成数据供训练、测试使用。

```
# 配置神经网络中间层的尺寸
input_size = X_train.shape[1] # input_size = 10
hidden1_size = 16
hidden2_size = 16
output_size = 4

# 初始化神经网络
nn_non_linear = ReluNeuralNetwork(input_size, hidden1_size, hidden2_size, output_size)
```

```
nn_linear = LinearNeuralNetwork(input_size, hidden1_size, hidden2_size, output_size)

# 训练神经网络
learning_rate = 0.1
nn_non_linear.train(X_train, y_train_onehot, epochs=300, learning_rate=learning_rate)
nn_linear.train(X_train, y_train_onehot, epochs=300, learning_rate=learning_rate)
```

这一部分代码配置了神经网络中中间层的尺寸,初始化神经网络,训练神经网络。

```
# 进行预测并评估准确率
predictions = nn_non_linear.forward(X_test)
predicted_classes = np.argmax(predictions, axis=1)
accuracy = np.mean(predicted_classes == y_test)
print(f"非线性模型测试准确率: {accuracy:.2f}")

predictions = nn_linear.forward(X_test)
predicted_classes = np.argmax(predictions, axis=1)
accuracy = np.mean(predicted_classes == y_test)
print(f"线性模型测试准确率: {accuracy:.2f}")
```

这一部分代码用来测试训练完成的神经网络的准确率。

1.5.3　具体任务:补全神经网络中前向传播和反向传播部分的代码

我们需要完成 ReluNeuralNetwork 和 LinearNeuralNetwork 这两个类中的 forward 和 backward 函数(共计四部分代码)。我们下面仅以 ReluNeuralNetwork 为例说明:

```
class ReluNeuralNetwork:
    def __init__(self, input_size, hidden1_size, hidden2_size, output_size):
        pass # 省略具体代码

    def forward(self, X):
        pass # 省略具体代码

    def backward(self, X, y, batch_size, y_pred, learning_rate):
        pass # 省略具体代码

    def train(self, X, y, epochs, learning_rate):
        pass # 省略具体代码
```

一个神经网络类中统共含有 __init__, forward, backward, train 四个函数。我们只需要补全 forward 和 backward 部分的代码。对于 forward 部分代码:

```
def forward(self, X):

    '''
    前向传播:第一隐藏层激活 a_1 的计算 [代码已经给出]
```

```
        '''
        self.z_1 = X @ self.weights_1 + self.bias_1
        self.a_1 = relu(self.z_1)

        '''
        TODO: 前向传播:第二隐藏层激活 a_2 的计算 [需要补全 None 代表部分的代码]
        '''
        self.z_2 = None
        self.a_2 = None

        '''
        TODO: 前向传播:输出层 y_pred 的计算 [需要补全 None 代表部分的代码]
        '''
        self.z_3 = None
        self.y_pred = softmax(self.z_3)

        return self.y_pred
```

参照实验原理部分补全 forward 中各层激活的计算代码。对于 backward 部分代码:

```
def backward(self, X, y, batch_size, y_pred, learning_rate):
        '''
        反向传播: self.weights_3, self.bias_3 参数梯度的计算和更新
        '''
        # 计算误差项 e_3
        e_3 = y_pred - y
        # self.weights_3, self.bias_3 参数梯度的计算
        grad_weights_3 = self.a_2.T @ e_3 / batch_size
        bias_weights_3 = np.sum(e_3, axis=0, keepdims=True) / batch_size
        # 梯度下降, 更新网络参数
        self.weights_3 -= grad_weights_3 * learning_rate
        self.bias_3 -= bias_weights_3 * learning_rate

        '''
        TODO: 反向传播: self.weights_1, self.bias_1 参数梯度的计算和更新
         [需要补全 None 代表部分的代码]
        '''
        # 计算误差项 e_2
        e_2 = None
        # self.weights_2, self.bias_2 参数梯度的计算
        grad_weights_2 = None
        grad_bias_2 = None
        # 梯度下降, 更新网络参数
        self.weights_2 -= None
```

```
        self.bias_2 -= None

        '''
        TODO: 反向传播: self.weights_2, self.bias_2 参数梯度的计算和更新
         [需要补全 None 代表部分的代码]
        '''
        # 计算误差项 e_1
        e_1 = None
        # self.weights_1, self.bias_1 参数梯度的计算
        grad_weights_1 = None
        grad_bias_1 = None
        # 梯度下降, 更新网络参数
        self.weights_1 -= None
        self.bias_1 -= None
```

参照实验原理部分补全 backward 中各层中误差项的计算、各个参数 (weights,biases) 梯度的计算和运用梯度更新算法更新参数。

1.6　实验要求与评分细则

1.6.1　实验要求

1. 完成 ReluNeuralNetwork 和 LinearNeuralNetwork 这两个类中的 forward 和 backward 函数 (共计四部分代码)

2. 成功运行数据线性可分和不可分两种情况下的实验, 并观察不同网络结构 (线性网络、非线性网络) 在处理不同数据分布时的效果差异性。

3. 提交时需要提交补全代码 (命名:bp_ 学号.py, 例:bp_PB24011035.py) 到 bb 系统, 并利用 python 中的注释功能, 在代码文件的末尾处写明自己运行结果和对不同网络结构 (线性网络、非线性网络) 在处理不同数据分布时的效果差异性的总结。

```
# 代码部分 ( 完整可运行代码 )

# 运行结果
'''
运行结果复制
'''

# 不同数据分布时的效果差异性的总结
'''
总结
'''
```

1.6.2 评分细则

评分细则主要依据以下几个方面进行评估：

（1）正确完成 ReluNeuralNetwork.forward() 部分代码的补全（20%）

（2）正确完成 ReluNeuralNetwork.backward() 部分代码的补全（20%）

（3）正确完成 LinearNeuralNetwork.forward() 部分代码的补全（20%）

（4）正确完成 LinearNeuralNetwork.forward() 部分代码的补全（20%）

（5）成功运行两种情况下的实验，总结不同网络结构（线性网络、非线性网络）在处理不同数据分布时的效果差异性（20%）

成功运行 python experiment_bp.py nonlinear 的结果示例：

```
# 非线性模型测试准确率：0.99
# 线性模型测试准确率：0.57
```

成功运行 python experiment_bp.py linear 的结果示例：

```
# 非线性模型测试准确率：1.00
# 线性模型测试准确率：1.00
```

1.7 思考题

不同网络结构（线性网络、非线性网络）在处理不同数据（线性可分、线性不可分）分布时的效果有何差异性？这种差异性由何而来？