第 2 章　神经网络设计实验

神经网络设计是设计复杂深度学习算法/应用的基础，本章将介绍如何设计一个三层神经网络模型来实现手写数字分类。首先介绍如何利用高级编程语言 Python 搭建神经网络训练和推断框架来实现手写数字分类的训练和使用。随后介绍如何移植到深度学习处理器 DLP 上实现手写数字分类。由于当前教学使用的 DLP 仅支持推断功能，因此本书中 DLP 相关实验仅实现神经网络推断功能。

2.1　基于三层神经网络实现手写数字分类

2.1.1　实验目的

掌握神经网络的设计原理，熟练掌握神经网络的训练和使用方法，能够使用 Python 语言实现一个三层全连接神经网络模型对手写数字分类的训练和使用。具体包括：

1) 实现三层神经网络模型进行手写数字分类，建立一个简单而完整的神经网络工程。通过本实验理解神经网络中基本模块的作用和模块间的关系，为后续建立更复杂的神经网络实验（如风格迁移）奠定基础。

2) 利用高级编程语言 Python 实现神经网络基本单元的前向传播（正向传播）和反向传播计算，加深对神经网络中基本单元的理解，包括全连接层、激活函数、损失函数等基本单元。

3) 利用高级编程语言 Python 实现神经网络构建，以及训练神经网络所使用的梯度下降算法，加深对神经网络训练过程的理解。

实验进程：5%。

实验工作量：约 200 行代码，约需 5 个小时。

2.1.2　背景知识

2.1.2.1　神经网络的组成

一个完整的神经网络通常由多个基本的网络层堆叠而成。本实验中的三层全连接神经网络由三个全连接层构成，在每两个全连接层之间会插入 ReLU 激活函数引入非线性变换，最后使用 Softmax 层计算交叉熵损失，如图2.1所示。因此本实验中使用的基本单元包括全连接层、ReLU 激活函数、Softmax 损失函数，在本节中将分别进行介绍。更多关于神经网络中基本单元的介绍详见《智能计算系统》教材[1] 第 2.3 节。

全连接层

全连接层以一维向量作为输入，输入与权重相乘后再与偏置相加得到输出向量。假设全连接层的输入为一维向量 $\boldsymbol{x}$，维度为 m，假设全连接层的输出为一维向量 $\boldsymbol{y}$，维度为 n。全连接层的权重 $\boldsymbol{W}$ 是二维矩阵，维度为 $m \times n$，偏置 $\boldsymbol{b}$ 是一维向量[①]，维度为 n。前向传播

[①]偏置可以是一维向量，计算每个输出使用不同的偏置值；偏置也可以是一个标量，计算同一层的输出使用同一个偏置值。

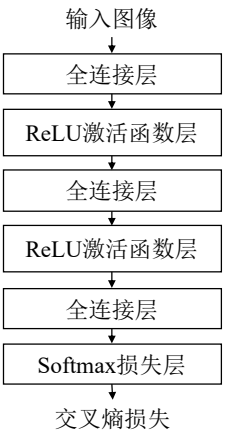

图 2.1　用于手写数字分类的三层全连接神经网络

时，全连接层的输出的计算公式为

$$\boldsymbol{y} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b} \tag{2.1}$$

在计算全连接层的反向传播时，给定神经网络损失函数 L 对当前全连接层的输出 $\boldsymbol{y}$ 的偏导 $\nabla_{\boldsymbol{y}}L = \frac{\partial L}{\partial \boldsymbol{y}}$，其维度与全连接层的输出 $\boldsymbol{y}$ 相同，均为 n。根据链式法则，全连接层的权重和偏置的梯度 $\nabla_{\boldsymbol{W}}L = \frac{\partial L}{\partial \boldsymbol{W}}$、$\nabla_{\boldsymbol{b}}L = \frac{\partial L}{\partial \boldsymbol{b}}$ 以及损失函数对输入的偏导 $\nabla_{\boldsymbol{x}}L = \frac{\partial L}{\partial \boldsymbol{x}}$ 计算公式为：

$$\begin{aligned}\nabla_{\boldsymbol{W}}L &= \boldsymbol{x}\nabla_{\boldsymbol{y}}L^T\\ \nabla_{\boldsymbol{b}}L &= \nabla_{\boldsymbol{y}}L\\ \nabla_{\boldsymbol{x}}L &= \boldsymbol{W}^T\nabla_{\boldsymbol{y}}L\end{aligned} \tag{2.2}$$

实际应用中通常使用批量随机梯度下降算法，即选择若干个样本同时计算。假设选择的样本量为 p，此时输入变为二维矩阵 $\boldsymbol{X}$，维度为 $p \times m$，每行代表一个样本。输出也变为二维矩阵 $\boldsymbol{Y}$，维度为 $p \times n$。此时全连接层的前向传播计算公式由公式(2.1)变为

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b} \tag{2.3}$$

其中的 + 代表广播运算，表示偏置 $\boldsymbol{b}$ 中的元素会被加到 $\boldsymbol{X}\boldsymbol{W}$ 的乘积矩阵的对应的一行元素中。权重和偏置的梯度 $\nabla_{\boldsymbol{W}}L$、$\nabla_{\boldsymbol{b}}L$ 以及损失函数对输入的偏导 $\nabla_{\boldsymbol{x}}L$ 的计算公式由公式(2.2)变为

$$\begin{aligned}\nabla_{\boldsymbol{W}}L &= \boldsymbol{X}^T\nabla_{\boldsymbol{Y}}L\\ \nabla_{\boldsymbol{b}}L &= \mathbf{1}\nabla_{\boldsymbol{Y}}L\\ \nabla_{\boldsymbol{X}}L &= \nabla_{\boldsymbol{Y}}L\boldsymbol{W}^T\end{aligned} \tag{2.4}$$

其中计算偏置的梯度 $\nabla_{\boldsymbol{b}}L$ 时，为确保维度正确，用 $\nabla_{\boldsymbol{Y}}L$ 与维度为 $1 \times p$ 的全 1 向量 $\mathbf{1}$ 相乘。

ReLU 激活函数

ReLU 激活函数是按元素运算操作，输出向量 $\boldsymbol{y}$ 的维度与输入向量 $\boldsymbol{x}$ 的维度相同[①]。在前向传播中，如果输入 $\boldsymbol{x}$ 中的元素小于 0，输出为 0，否则输出等于输入。因此 ReLU 的计

[①]输入和输出也可以是矩阵，处理方式类似。

算公式为

$$\boldsymbol{y}(i) = \max(0, \boldsymbol{x}(i)) \tag{2.5}$$

其中 $\boldsymbol{x}(i)$ 和 $\boldsymbol{y}(i)$ 分别代表 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 在位置 i 的值。

由于 ReLU 激活函数不包含参数，在反向传播计算过程中仅需根据损失函数对输出的偏导 $\nabla_{\boldsymbol{y}} L$ 计算损失函数对输入的偏导 $\nabla_{\boldsymbol{x}} L$。设 i 代表输入 $\boldsymbol{x}$ 的某个位置，则损失函数对本层的第 i 个输入的偏导 $\nabla_{\boldsymbol{x}(i)} L$ 的计算公式为

$$\nabla_{\boldsymbol{x}(i)} L = \begin{cases} \nabla_{\boldsymbol{y}(i)} L, & \boldsymbol{x}(i) \geq 0 \\ 0, & \boldsymbol{x}(i) < 0 \end{cases} \tag{2.6}$$

Softmax 损失层

Softmax 损失层是目前多分类问题中最常用的损失函数层。假设 Softmax 损失层的输入为向量 $\boldsymbol{x}$，维度为 k。其中 k 对应分类的类别数，如对手写数字 0 至 9 进行分类时，类别数 $k = 10$。在前向传播的计算过程中，首先对 $\boldsymbol{x}$ 计算 e 指数并进行行归一化，从而得到 Softmax 分类概率。假设 $\boldsymbol{x}$ 对应 i 位置的值为 $\boldsymbol{x}(i)$，$\hat{\boldsymbol{x}}(i)$ 为 i 位置的 Softmax 分类概率，$i \in [1, k]$ 且为整数，则 $\hat{\boldsymbol{y}}(i)$ 的计算公式为

$$\hat{\boldsymbol{y}}(i) = \frac{e^{\boldsymbol{x}(i)}}{\sum\limits_j e^{\boldsymbol{x}(j)}} \tag{2.7}$$

在前向计算时，对 Softmax 分类概率 $\hat{\boldsymbol{y}}$ 取最大概率对应的类别作为预测的分类类别。损失函数层在计算前向传播时还需要根据给定的标记（label，也称为真实值或实际值）$\boldsymbol{y}$ 计算总的损失函数值。在分类任务中，标记 $\boldsymbol{y}$ 通常表示为一个维度为 k 的 one-hot 向量，该向量中对应真实类别的分量值为 1，其他值为 0。Softmax 损失层使用了交叉熵计算损失值，其损失值 L 的计算公式为

$$L = -\sum_i \boldsymbol{y}(i) \ln \hat{\boldsymbol{y}}(i) \tag{2.8}$$

在反向传播的计算过程中，可直接利用标记数据和损失函数层的输出计算本层输入的损失。对于 Softmax 损失函数层，损失函数对输入的偏导 $\nabla_{\boldsymbol{x}} L$ 的计算公式为

$$\nabla_{\boldsymbol{x}} L = \frac{\partial L}{\partial \boldsymbol{x}} = \hat{\boldsymbol{y}} - \boldsymbol{y} \tag{2.9}$$

由于工程实现中使用批量随机梯度下降算法，假设选择的样本量为 p，Softmax 损失层的输入变为二维矩阵 $\boldsymbol{X}$，维度为 $p \times k$，$\boldsymbol{X}$ 的每个行向量代表一个样本。则对每个样本的激活值计算 e 指数并进行行归一化得到

$$\hat{\boldsymbol{Y}}(i,j) = \frac{e^{\boldsymbol{X}(i,j)}}{\sum\limits_j e^{\boldsymbol{X}(i,j)}} \tag{2.10}$$

其中 $\boldsymbol{X}(i,j)$ 代表 $\boldsymbol{X}$ 中对应第 i 样本 j 位置的值。当 $\boldsymbol{X}(i,j)$ 数值较大时，求 e 指数可能会出现数值上溢的问题。因此在实际工程实现时，为确保数值稳定性，会在求 e 指数前先进行减最大值处理，此时 $\hat{\boldsymbol{Y}}(i,j)$ 的计算公式变为

$$\hat{\boldsymbol{Y}}(i,j) = \frac{e^{\boldsymbol{X}(i,j) - \max\limits_n \boldsymbol{X}(i,n)}}{\sum\limits_j e^{\boldsymbol{X}(i,j) - \max\limits_n \boldsymbol{X}(i,n)}} \tag{2.11}$$

在前向计算时，对 Softmax 分类概率 $\hat{\boldsymbol{Y}}(i,j)$ 的每个样本（即每个行向量）取最大概率对应的类别作为预测的分类类别。此时标记 $\boldsymbol{Y}$ 通常表示为一组 one-hot 向量，维度为 $p \times k$，其中每行是一个 one-hot 向量，对应一个样本的标记。则计算损失值的公式(2.8)变为

$$L = -\frac{1}{p}\sum_{i,j}\boldsymbol{Y}(i,j)\ln\hat{\boldsymbol{Y}}(i,j) \tag{2.12}$$

其中损失值是所有样本的平均损失，因此对样本数量 p 取平均。

在反向传播时，当选择的样本量为 p 时，损失函数对输入的偏导 $\nabla_{\boldsymbol{X}}L$ 的计算公式(2.9)变为：

$$\nabla_{\boldsymbol{X}}L = \frac{1}{p}(\hat{\boldsymbol{Y}} - \boldsymbol{Y}) \tag{2.13}$$

类似地，损失 $\nabla_{\boldsymbol{X}}L$ 是所有样本的平均损失，因此对样本数量 p 取平均。

2.1.2.2 神经网络训练

神经网络训练通过调整网络层的参数来使神经网络计算出来的结果与真实结果（标记）尽量接近。神经网络训练通常使用随机梯度下降算法，通过不断的迭代计算每层参数的梯度，利用梯度对每层参数进行更新。具体而言，给定当前迭代的训练样本（包含输入数据及标记信息），首先进行神经网络的前向传播处理，输入数据和权重相乘再经过激活函数计算出隐层，隐层与下一层的权重相乘再经过激活函数得到下一个隐层，通过逐层迭代计算出神经网络的输出结果。随后利用输出结果和标记信息计算出损失函数值。然后进行神经网络的反向传播处理，从损失函数开始逆序逐层计算损失函数对权重和偏置的偏导（即梯度），最后利用梯度对相应的参数进行更新。更新参数 $\boldsymbol{W}$ 的计算公式为

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta\nabla_{\boldsymbol{W}}L \tag{2.14}$$

其中，$\nabla_{\boldsymbol{W}}L$ 为参数的梯度，η 是学习率。

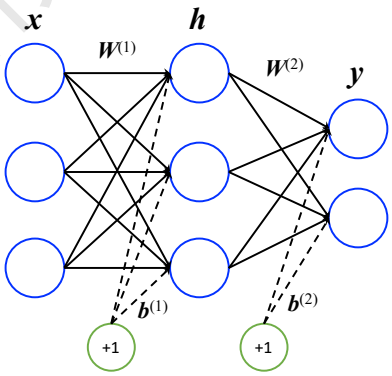

图 2.2　两层神经网络示例

下面以图2.2中的两层神经网络为例，介绍神经网络训练的具体过程。图2.2中的网络由两个全连接层及 Softmax 层组成[①]，其中第一个全连接层的权重为 $\boldsymbol{W}^{(1)}$，偏置为 $\boldsymbol{b}^{(1)}$，第二

[①]本实验中实现的三层神经网络与这个例子的非常类似，在这个例子基础上再添加一层全连接层和一层 ReLU 层即可，网络训练的过程也与这个例子完全一致

个全连接层的权重为 $\boldsymbol{W}^{(2)}$，偏置为 $\boldsymbol{b}^{(2)}$。假设某次迭代的网络输入为 $\boldsymbol{x}$，对应的标记为 $\boldsymbol{y}$。该神经网络前向传播的逐层计算公式依次是

$$\begin{aligned} \boldsymbol{h} &= \boldsymbol{W}^{(1)^T}\boldsymbol{x} + \boldsymbol{b}^{(1)} \\ \boldsymbol{z} &= \boldsymbol{W}^{(2)^T}\boldsymbol{h} + \boldsymbol{b}^{(2)} \end{aligned} \tag{2.15}$$

其中 $\boldsymbol{h}, \boldsymbol{z}$ 分别是第一、第二层全连接层的输出。Softmax 损失的损失值 L 为：

$$\begin{aligned} \hat{\boldsymbol{y}}(i) &= \frac{e^{\boldsymbol{z}(i)}}{\sum_i e^{\boldsymbol{z}(i)}} \\ L &= -\sum_i \boldsymbol{y}(i) \ln \hat{\boldsymbol{y}}(i) \end{aligned} \tag{2.16}$$

反向传播的逐层计算公式为

$$\begin{aligned} \nabla_{\boldsymbol{z}} L &= \hat{\boldsymbol{y}} - \boldsymbol{y} \\ \nabla_{\boldsymbol{W}^{(2)}} L &= \boldsymbol{h} \nabla_{\boldsymbol{z}} L^T \\ \nabla_{\boldsymbol{b}^{(2)}} L &= \nabla_{\boldsymbol{z}} L \\ \nabla_{\boldsymbol{h}} L &= \boldsymbol{W}^{(2)^T} \nabla_{\boldsymbol{z}} L \\ \nabla_{\boldsymbol{W}^{(1)}} L &= x \nabla_{\boldsymbol{h}} L^T \\ \nabla_{\boldsymbol{b}^{(1)}} L &= \nabla_{\boldsymbol{h}} L \\ \nabla_{\boldsymbol{x}} L &= \boldsymbol{W}^{(1)^T} \nabla_{\boldsymbol{h}} L \end{aligned} \tag{2.17}$$

其中 $\nabla_{\boldsymbol{z}} L$、$\nabla_{\boldsymbol{h}} L$、$\nabla_{\boldsymbol{x}} L$ 分别是损失函数对 Softmax 层、第二层、第一层的偏导，$\nabla_{\boldsymbol{W}^{(2)}} L$、$\nabla_{\boldsymbol{b}^{(2)}} L$、$\nabla_{\boldsymbol{W}^{(1)}} L$、$\nabla_{\boldsymbol{b}^{(1)}} L$ 分别是第二层和第一层的权重和偏置梯度，η 为学习率。更新两个全连接层的权重和偏置的计算为

$$\begin{aligned} \boldsymbol{W}^{(1)} &\leftarrow \boldsymbol{W}^{(1)} - \eta \nabla_{\boldsymbol{W}^{(1)}} L \\ \boldsymbol{b}^{(1)} &\leftarrow \boldsymbol{b}^{(1)} - \eta \nabla_{\boldsymbol{b}^{(1)}} L \\ \boldsymbol{W}^{(2)} &\leftarrow \boldsymbol{W}^{(2)} - \eta \nabla_{\boldsymbol{W}^{(2)}} L \\ \boldsymbol{b}^{(2)} &\leftarrow \boldsymbol{b}^{(2)} - \eta \nabla_{\boldsymbol{b}^{(2)}} L \end{aligned} \tag{2.18}$$

神经网络训练相关的详细介绍可以参见《智能计算系统》教材第 2.2 节。

2.1.3 实验环境

硬件环境：CPU。

软件环境：Python 编译环境及相关的扩展库，包括 Python 2.7.9，Pillow 3.4.2，SciPy 0.18.1，NumPy 1.11.2（本实验不需使用 TensorFlow 等深度学习框架）。

数据集：MNIST 手写数字库[2]。该数据集包含一个训练集和一个测试集，其中训练集有 60000 个样本，测试集有 10000 个样本。每个样本都由灰度图像（即单通道图像）及其标记组成，图像大小为 28×28。MNIST 数据集包含 4 个文件，分别是训练集图像、训练集标记、测试集图像、测试集标记。下载地址为http://yann.lecun.com/exdb/mnist/。

2.1.4 实验内容

设计一个三层神经网络实现手写数字图像分类。该网络包含两个隐层和一个输出层，其中输入神经元个数由输入数据维度决定，输出层的神经元个数由数据集包含的类别决定，

两个隐层的神经元个数可以作为超参数自行设置。对于手写数字图像的分类问题，输入数据为手写数字图像，原始图像一般可表示为二维矩阵（灰度图像）或三维矩阵（彩色图像），在输入神经网络前会将图像矩阵调整为一维向量作为输入。待分类的类别数一般是提前预设的，如手写数字包含 0 至 9 共 10 个类别，则神经网络的输出神经元个数为 10。

为了便于迭代开发，工程实现时采用模块化的方式来实现整个神经网络的处理。目前绝大多数神经网络工程实现时通常划分为 5 大模块：

1) 数据加载模块：从文件中读取数据，并进行预处理，其中预处理包括归一化、维度变换等处理。如果需要人为对数据进行随机数据扩增，则数据扩增处理也在数据加载模块中实现。

2) 基本单元模块：实现神经网络中不同类型的网络层的定义、前向传播计算、反向传播计算等功能。

3) 网络结构模块：利用基本单元模块建立一个完整的神经网络。

4) 网络训练（training）模块：该模块实现用训练集进行神经网络训练的功能。在已建立的神经网络结构基础上，实现神经网络的前向传播、神经网络的反向传播、对神经网络进行参数更新、保存神经网络参数等基本操作，以及训练函数主体。

5) 网络推断（inference）模块：该模块实现使用训练得到的网络模型，对测试样本进行预测的过程[①]。具体实现的操作包括训练得到的模型参数的加载、神经网络的前向传播等。

在某些特殊应用场景下，神经网络工程可能不需要包含所有模块，例如仅实现推断过程的工程中通常不包含训练模块（如实验3.1），非实时的风格迁移工程中不包含推断模块（如实验3.3）。

本实验采用上述较为细致的模块划分方式。目前网络上有些开源的神经网络工程可能采用较粗的模块划分方式，例如将基本单元模块与网络结构模块合并，或将网络训练模块与网络推断模块合并。

2.1.5 实验步骤

本节介绍如何实现本实验涉及的各个模块，以及如何搭建和调用各个模块来实现手写数字图像分类。

2.1.5.1 数据加载模块

本实验采用的数据集是 MNIST 手写数字库[2]。该数据集中的图像数据和标记数据采用表2.1中的 IDX 文件格式存放。图像的像素值按行优先顺序存放，取值范围为 [0,255]，其中 0 表示黑色，255 表示白色。

首先编写读取 MNIST 数据集文件并预处理的子函数，程序示例如图2.3所示。然后调用该子函数对 MNIST 数据集中的 4 个文件分别进行读取和预处理，并将处理过的训练和测试数据存储在 NumPy 矩阵中（训练模型时可以快速读取该矩阵中的数据），实现该功能的程序示例如图2.4所示。

[①]在不同文献中可能被称为测试、推断或预测。

表 2.1　MNIST 数据集 IDX 文件格式[2]

图像文件格式			
字节偏移	数据类型	值	描述
0000	int32（32 位有符号整型）	0x00000803(2051)	magic number（魔数）：表示像素的数据类型以及像素数据的维度信息，MSB（大尾端）
0004	int32	60000（训练集） 10000（测试集）	图像数量
0008	int32	28	图像行数，即图像高度
0012	int32	28	图像列数，即图像宽度
0016	uint8（8 位无符号整型）	??	像素值
0017	uint8	??	像素值
.....			
xxxx	uint8	??	像素值
标记文件格式			
字节偏移	数据类型	值	描述
0000	int32	0x00000801(2049)	magic number
0004	int32	60000（训练集） 10000（测试集）	标记数量
0008	uint8	??	像素值
0009	uint8	??	像素值
.....			
xxxx	uint8	??	像素值

```python
def load_mnist(self, file_dir, is_images = 'True'):
    bin_file = open(file_dir, 'rb')
    bin_data = bin_file.read()
    bin_file.close()
    if is_images: # 读取图像数据
        fmt_header = '>iiii'
        magic, num_images, num_rows, num_cols = struct.unpack_from(fmt_header, bin_data,
    0)
    else: # 读取标记数据
        fmt_header = '>ii'
        magic, num_images = struct.unpack_from(fmt_header, bin_data, 0)
        num_rows, num_cols = 1, 1
    data_size = num_images * num_rows * num_cols
    mat_data = struct.unpack_from('>' + str(data_size) + 'B', bin_data, struct.calcsize(
    fmt_header))
    mat_data = np.reshape(mat_data, [num_images, num_rows * num_cols])
    return mat_data
```

图 2.3　MNIST 数据集文件的读取和预处理

```python
def load_data(self):
    # TODO: 调用函数load_mnist读取和预处理MNIST中训练数据和测试数据的图像和标记
    train_images = self.load_mnist(os.path.join(MNIST_DIR, TRAIN_DATA), True)
    train_labels = _____________
    test_images = _____________
    test_labels = _____________
    self.train_data = np.append(train_images, train_labels, axis=1)
    self.test_data = np.append(test_images, test_labels, axis=1)
```

图 2.4　MNIST 子数据集的读取和预处理

2.1.5.2 基本单元模块

本实验采用图2.1中的三层神经网络，主体是三个全连接层。在前两个全连接层之后使用 ReLU 激活函数层引入非线性变换，本实验采用 ReLU 层作为激活函数层。在神经网络的最后添加 Softmax 层计算交叉熵损失。因此，本实验中需要实现的基本单元模块包括全连接层、ReLU 层和 Softmax 损失层。

在神经网络实现中，通常同类型的层用一个类来定义，多个同类型的层用类的实例来实现，层中的计算用类的成员函数来定义。类的成员函数通常包括层的初始化、参数的初始化、前向传播计算、反向传播计算、参数的更新、参数的加载和保存等。其中层的初始化函数一般会根据实例化层时的输入系数确定该层的超参数，例如该层的输入神经元数量和输出神经元数量等。参数的初始化函数会对该层的参数（如全连接层中的权重和偏置）分配存储空间，并填充初始值。前向传播函数利用前一层的输出作为本层的输入，计算本层的输出结果。反向传播函数根据链式法则逆序逐层计算损失函数对权重和偏置的梯度。参数的更新函数利用反向传播函数计算的梯度对本层的参数进行更新。参数的加载函数从给定的文件中加载参数的值，参数的保存函数将当前层参数的值保持到指定的文件中。有些层（如激活函数层）可能没有参数，就不需要定义参数的初始化、更新、加载和保存函数。有些层（如激活函数层和损失函数层）的输出维度由输入维度决定，不需要人工设定，因此不需要层的初始化函数。

以下是是全连接层、ReLU 层和 Softmax 损失层的具体实现步骤。

全连接层：程序示例如图2.5所示，定义了以下成员函数。

• 层的初始化：需要确定该全连接层的输入神经元个数（即输入二维矩阵中每个行向量的维度）和输出神经元个数（即输出二维矩阵中每个行向量的维度）。

• 参数初始化：全连接层的参数包括权重和偏置。根据输入向量的维度 m 和输出向量的维度 n 可以确定权重 $\boldsymbol{W}$ 的维度为 $m \times n$，偏置 $\boldsymbol{b}$ 的维度为 n。在对权重和偏置进行初始化时，通常利用高斯随机数初始化权重的值，而将偏置的所有值初始化为 0。

• 前向传播计算：全连接层的前向传播计算公式为(2.3)，可以通过输入矩阵与权重矩阵相乘再与偏置相加实现。

• 反向传播计算：全连接层的反向传播的计算公式为(2.4)。给定损失函数对本层输出的偏导 $\nabla_{\boldsymbol{y}} L$，利用矩阵相乘计算权重和偏置的梯度 $\nabla_{\boldsymbol{W}} L$、$\nabla_{\boldsymbol{b}} L$ 以及损失函数对本层输入的偏导 $\nabla_{\boldsymbol{x}} L$。

• 参数更新：给定学习率 η，可利用反向传播计算得到的权重梯度 $\nabla_{\boldsymbol{W}} L$ 和偏置梯度 $\nabla_{\boldsymbol{b}} L$ 对本层的权重 $\boldsymbol{W}$ 和偏置 $\boldsymbol{b}$ 进行更新：

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \nabla_{\boldsymbol{W}} L \tag{2.19}$$

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \eta \nabla_{\boldsymbol{b}} L \tag{2.20}$$

• 参数加载：从该函数的输入中读取本层的权重 $\boldsymbol{W}$ 和偏置 $\boldsymbol{b}$。

• 参数保存：返回本层当前的权重 $\boldsymbol{W}$ 和偏置 $\boldsymbol{b}$。

ReLU 层：不包含参数，因此实现中没有参数初始化、参数更新、参数的加载和保存相关的函数。ReLU 层的程序示例如图2.6所示，定义了以下成员函数。

```python
class FullyConnectedLayer(object):
    def __init__(self, num_input, num_output): # 全连接层初始化
        self.num_input = num_input
        self.num_output = num_output
    def init_param(self, std=0.01): # 参数初始化
        self.weight = np.random.normal(loc=0.0, scale=std, size=(self.num_input, self.
     num_output))
        self.bias = np.zeros([1, self.num_output])
    def forward(self, input): # 前向传播的计算
        self.input = input
        # TODO: 全连接层的前向传播，计算输出结果
        self.output = _____________
        return self.output
    def backward(self, top_diff): # 反向传播的计算
        # TODO: 全连接层的反向传播，计算参数梯度和本层损失
        self.d_weight = _____________
        self.d_bias = _____________
        bottom_diff = _____________
        return bottom_diff
    def update_param(self, lr): # 参数更新
        # TODO: 对全连接层参数利用梯度进行更新
        self.weight = _____________
        self.bias = _____________
    def load_param(self, weight, bias): # 参数加载
        self.weight = weight
        self.bias = bias
    def save_param(self): # 参数保存
        return self.weight, self.bias
```

图 2.5　全连接层的实现示例

• 前向传播计算：根据公式(2.5)可以计算 ReLU 层前向传播的结果。在工程实现中，可以对整个输入矩阵使用 maximum 函数，maximum 函数会进行广播，计算输入矩阵的每个元素与 0 的最大值。

• 反向传播计算：根据公式(2.6)可以计算损失函数对输入的偏导。在工程实现中，可以获取 $x(i) < 0$ 的位置索引，将 $\boldsymbol{y}$ 中对应位置的值置为 0。

```python
class ReLULayer(object):
    def forward(self, input): # 前向传播的计算
        self.input = input
        # TODO: ReLU层的前向传播，计算输出结果
        output = _____________
        return output
    def backward(self, top_diff): # 反向传播的计算
        # TODO: ReLU层的反向传播，计算本层损失
        bottom_diff = _____________
        return bottom_diff
```

图 2.6　ReLU 层的实现示例

Softmax 损失层：同样不包含参数，因此实现中没有参数初始化、更新、加载和保存相关的函数。但作为该层需要额外计算总的损失函数值，作为训练时的中间输出结果，帮助判断模型的训练进程。Softmax 损失层的程序示例如图2.7所示，定义了以下成员函数。

• 前向传播计算：可以使用公式(2.11)计算，该公式为确保数值稳定，会在求 e 指数前先进行减最大值处理。

• 损失函数计算：可以使用公式(2.12)计算，采用批量随机梯度下降法训练时损失值是 batch 内的所有样本的损失值的均值。需要注意的是，MNIST 手写数字库的标记数据读入的是 0 至 9 的类别编号，在计算损失时需要先将类别编号转换为 one-hot 向量。

• 反向传播计算：可以使用公式(2.13)计算，计算时同样需要对样本数量取平均。

```
class SoftmaxLossLayer(object):
    def forward(self, input): # 前向传播的计算
        # TODO: softmax损失层的前向传播，计算输出结果
        input_max = np.max(input, axis=1, keepdims=True)
        input_exp = np.exp(input - input_max)
        self.prob = _____________
        return self.prob
    def get_loss(self, label): # 计算损失
        self.batch_size = self.prob.shape[0]
        self.label_onehot = np.zeros_like(self.prob)
        self.label_onehot[np.arange(self.batch_size), label] = 1.0
        loss = -np.sum(np.log(self.prob) * self.label_onehot) / self.batch_size
        return loss
    def backward(self): # 反向传播的计算
        # TODO: softmax损失层的反向传播，计算本层损失
        bottom_diff = _____________
        return bottom_diff
```

图 2.7　Softmax 损失层的实现示例

2.1.5.3　网络结构模块

网络结构模块利用已经实现的神经网络的基本单元来建立一个完整的神经网络。在工程实现中通常用一个类来定义一个神经网络，用类的成员函数来定义神经网络的初始化、建立神经网络结构、对神经网络进行参数初始化等基本操作。本实验中三层神经网络的网络结构模块的程序示例如图2.8所示，定义了以下成员函数。

• 神经网络初始化：确定神经网络相关的超参数，例如网络中每个隐层的神经元个数。

• 建立网络结构：定义整个神经网络的拓扑结构，实例化基本单元模块中定义的层并将这些层进行堆叠。例如本实验使用的三层神经网络包含三个全连接层，并且在前两个全连接层后跟随有 ReLU 层，神经网络的最后使用了 Softmax 损失层。

• 神经网络参数初始化：对于神经网络中包含参数的层，依次调用这些层的参数初始化函数，从而完成整个神经网络的参数初始化。本实验使用的三层神经网络中，只有三个全连接层包含参数，依次调用其参数初始化函数即可。

2.1.5.4　网络训练模块

神经网络训练流程如图2.9所示。在完成数据加载模块和网络结构模块实现之后，需要实现训练模块。本实验中三层神经网络的网络训练模块程序示例如图2.10所示。神经网络的训练模块通常拆解为若干步骤，包括神经网络的前向传播、神经网络的反向传播、神经网络参数更新、神经网络参数保存等基本操作。这些网络训练模块的基本操作以及训练主体用神经网络类的成员函数来定义：

• 神经网络的前向传播：根据神经网络的拓扑顺序，顺序调用每层的前向传播函数。以

```python
class MNIST_MLP(object):
    def __init__(self, batch_size=100, input_size=784, hidden1=32, hidden2=16,
     out_classes=10, lr=0.01, max_epoch=2, print_iter=100):
        # 神经网络初始化
        self.batch_size = batch_size
        self.input_size = input_size
        self.hidden1 = hidden1
        self.hidden2 = hidden2
        self.out_classes = out_classes
        self.lr = lr
        self.max_epoch = max_epoch
        self.print_iter = print_iter
    def build_model(self): # 建立网络结构
        # TODO: 建立三层神经网络结构
        self.fc1 = FullyConnectedLayer(self.input_size, self.hidden1)
        self.relu1 = ReLULayer()
        ____________________________________
        self.fc3 = FullyConnectedLayer(self.hidden2, self.out_classes)
        self.softmax = SoftmaxLossLayer()
        self.update_layer_list = [self.fc1, self.fc2, self.fc3]
    def init_model(self): # 神经网络参数初始化
        for layer in self.update_layer_list:
            layer.init_param()
```

图 2.8　三层神经网络的网络结构模块实现示例

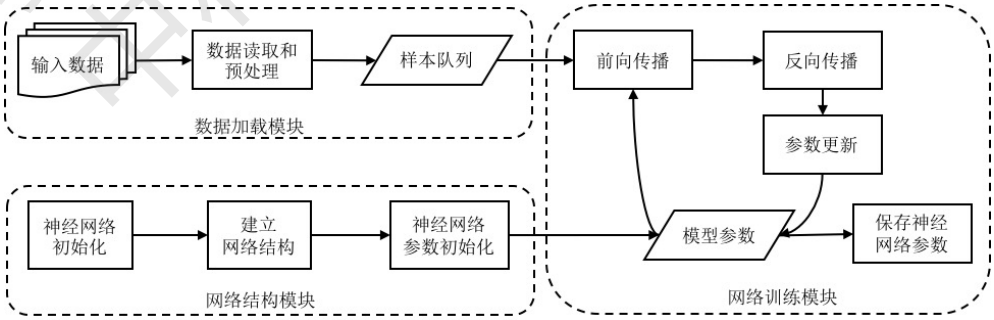

图 2.9　神经网络训练流程

输入数据作为第一层的输入，之后每层的输出作为其后一层的输入，顺序计算每一层的输出，最后得到损失函数层的输出。

• 神经网络的反向传播：根据神经网络的拓扑顺序，逆序调用每层的反向传播函数。采用链式法则逆序逐层计算损失函数对每层参数的偏导，最后得到神经网络所有层的参数梯度。

• 神经网络参数更新：对神经网络中包含参数的层，依次调用各层的参数更新函数，来对整个神经网络的参数进行更新。本实验中的三层神经网络仅其中的三个全连接层包含参数，依次更新三个全连接层的参数即可。

• 神经网络参数保存：对神经网络中包含参数的层，依次收集这些层的参数并存储到文件中。

• 神经网络训练主体：在该函数中，（1）确定训练的一些超参数，如使用批量梯度下降算法时的批量大小、学习率大小、迭代次数（或训练周期次数）、可视化训练过程时每迭代多少次屏幕输出一次当前的损失值等等。（2）开始迭代训练过程。每次迭代训练开始前，可以根据需要对数据进行随机打乱，一般是一个训练周期（即当整个数据集的数据都参与一次训练过程）后对数据集进行随机打乱。每次迭代训练过程中，先选取当前迭代所使用的数据和对应的标记，再进行整个网络的前向传播，随后计算当前迭代的损失值，然后进行整个网络的反向传播来获得整个网络的参数梯度，最后对整个网络的参数进行更新。完成一次迭代后可以根据需要在屏幕上输出当前的损失值，以供实际应用中修改模型作参考。完成神经网络的训练过程后，通常会将训练得到的神经网络模型参数保存到文件中。

2.1.5.5 网络推断模块

整个神经网络推断流程如图2.11所示。完成神经网络的训练之后，可以用训练得到的模型对测试数据进行预测，以评估模型的精度。本实验中三层神经网络的网络推断模块程序示例如图2.12所示。工程实现中同样常将一个神经网络的推断模块拆解为若干步骤，包括神经网络模型参数加载、前向传播、精度计算等基本操作。这些网络推断模块的基本操作以及推断主体用神经网络类的成员函数来定义：

• 神经网络的前向传播：网络推断模块中的神经网络前向传播操作与网络训练模块中的前向传播操作完全一致，因此可以直接调用网络训练模块中的神经网络前向传播函数。

• 神经网络参数加载：读取神经网络训练模块保存的模型参数文件，并加载有参数的网络层的参数值。

• 神经网络推断函数主体：在进行神经网络推断前，需要从模型参数文件中加载神经网络的参数。在神经网络推断过程中，循环每次读取一定批量的测试数据，随后进行整个神经网络的前向传播计算得到神经网络的输出结果。得到整个测试数据集的输出结果后，与测试数据集的标记进行比对，利用相关的评价函数计算模型的精度，如手写数字分类问题使用分类平均正确率作为模型的评价函数。

2.1.5.6 完整实验流程

完成神经网络的各个模块之后，调用这些模块就可以实现用三层神经网络进行手写数字图像分类的完整流程。本实验中三层神经网络的完整流程的程序示例如图2.13所示。首

```python
def forward(self, input): # 神经网络的前向传播
    # TODO: 神经网络的前向传播
    h1 = self.fc1.forward(input)
    h1 = self.relu1.forward(h1)
    ______________
    prob = self.softmax.forward(h3)
    return prob

def backward(self): # 神经网络的反向传播
    # TODO: 神经网络的反向传播
    dloss = self.softmax.backward()
    ______________
    dh1 = self.relu1.backward(dh2)
    dh1 = self.fc1.backward(dh1)

def update(self, lr): # 神经网络参数更新
    for layer in self.update_layer_list:
        layer.update_param(lr)

def save_model(self, param_dir): # 保存神经网络参数
    params = {}
    params['w1'], params['b1'] = self.fc1.save_param()
    params['w2'], params['b2'] = self.fc2.save_param()
    params['w3'], params['b3'] = self.fc3.save_param()
    np.save(param_dir, params)

def train(self): # 训练函数主体
    max_batch = self.train_data.shape[0] / self.batch_size
    for idx_epoch in range(self.max_epoch):
        mlp.shuffle_data()
        for idx_batch in range(max_batch):
            batch_images = self.train_data[idx_batch*self.batch_size:(idx_batch+1)*self.batch_size, :-1]
            batch_labels = self.train_data[idx_batch*self.batch_size:(idx_batch+1)*self.batch_size, -1]
            prob = self.forward(batch_images)
            loss = self.softmax.get_loss(batch_labels)
            self.backward()
            self.update(self.lr)
            if idx_batch % self.print_iter == 0:
                print('Epoch %d, iter %d, loss: %.6f' % (idx_epoch, idx_batch, loss))
```

图 2.10　三层神经网络的网络训练模块实现示例

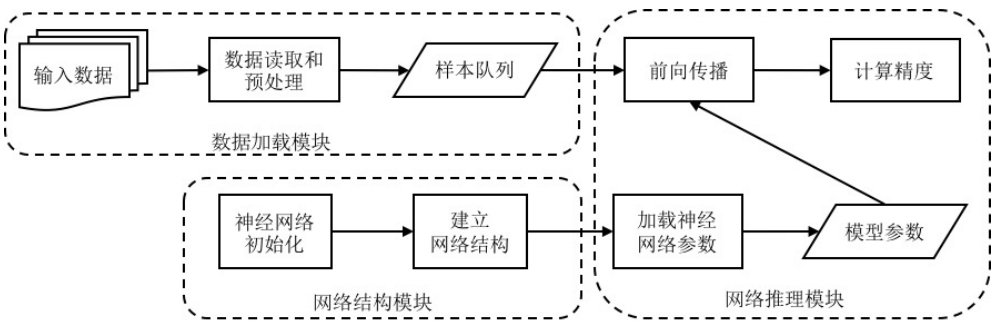

图 2.11　神经网络推断流程

```
def load_model(self, param_dir): # 加载神经网络参数
    params = np.load(param_dir).item()
    self.fc1.load_param(params['w1'], params['b1'])
    self.fc2.load_param(params['w2'], params['b2'])
    self.fc3.load_param(params['w3'], params['b3'])

def evaluate(self): # 推断函数主体
    pred_results = np.zeros([self.test_data.shape[0]])
    for idx in range(self.test_data.shape[0]/self.batch_size):
        batch_images = self.test_data[idx*self.batch_size:(idx+1)*self.batch_size, :-1]
        prob = self.forward(batch_images)
        pred_labels = np.argmax(prob, axis=1)
        pred_results[idx*self.batch_size:(idx+1)*self.batch_size] = pred_labels
    accuracy = np.mean(pred_results == self.test_data[:,-1])
    print('Accuracy in test set: %f' % accuracy)
```

图 2.12　三层神经网络的网络推断模块实现示例

先实例化三层神经网络对应的类，指定神经网络的超参数，如每层的神经元个数。其次进行数据的加载和预处理。再调用网络结构模块建立神经网络，随后进行网络初始化，在该过程中网络结构模块会自动调用基本单元模块实例化神经网络中的每个层。然后调用网络训练模块训练整个网络，之后将训练得到的模型参数保存到文件中。最后从文件中读取训练得到的模型参数，之后调用网络推断模块测试网络的精度。

```
if __name__ == '__main__':
    h1, h2, e = 32, 16, 1
    mlp = MNIST_MLP(hidden1=h1, hidden2=h2, max_epoch=e)
    mlp.load_data()
    mlp.build_model()
    mlp.init_model()
    mlp.train()
    mlp.save_model('mlp-%d-%d-%depoch.npy' % (h1, h2, e))
    mlp.load_model('mlp-%d-%d-%depoch.npy' % (h1, h2, e))
    mlp.evaluate()
```

图 2.13　三层神经网络的完整流程实现示例

2.1.6　实验评估

在图像分类任务中，通常使用测试集的平均分类正确率判断分类结果的精度。假设共有 N 个图像样本（MNIST 手写数据集中共包含 10000 张测试图像，此时 $N = 10000$），bmp_i 为神经网络输出的第 i 张图像的预测结果，$\boldsymbol{p}_i$ 为一个向量，取其中最大分量对应的类别作为预测类别。假设第 i 张图像的标记为 y_i，即第 i 张图像属于类别 y_i，则计算平均分类正确率 R 的公式为

$$R = \frac{1}{N}\sum_{i=1}^{N}\mathbf{1}(\text{argmax}(\boldsymbol{p}_i) = y_i) \tag{2.21}$$

其中 $\mathbf{1}(\text{argmax}(p_i) = y_i)$ 代表当 $\boldsymbol{p}_i$ 中的最大分量对应的类别编号与 y_i 相等时值为 1，否则值为 0。

本实验的评估标准设定如下：

• 60 分标准：给定全连接层、ReLU 层、Softmax 损失层的前向传播的输入矩阵、参数值、反向传播的输入，可以得到正确的前向传播的输出矩阵、反向传播的输出和参数梯度。

• 80 分标准：实现正确的三层神经网络，并进行训练和推断，使最后训练得到的模型在 MNIST 测试数据集上的平均分类正确率高于 92%。

• 90 分标准：实现正确的三层神经网络，并进行训练和推断，调整和训练相关的超参数，使最后训练得到的模型在 MNIST 测试数据集上的平均分类正确率高于 95%。

• 100 分标准：在三层神经网络基础上设计自己的神经网络结构，并进行训练和推断，使最后训练得到的模型在 MNIST 测试数据集上的平均分类正确率高于 98%。

2.1.7 实验思考

在实验中请思考如下问题：

1) 在实现神经网络基本单元时，如何确保一个层的实现是正确的?

2) 在实现神经网络后，如何在不改变网络结构的条件下提高精度?

3) 如何通过修改网络结构提高精度？可以从哪些方面修改网络结构?

2.2 基于 DLP 平台实现手写数字分类

2.2.1 实验目的

熟悉深度学习处理器 DLP 平台的使用，能使用已封装好的 Python 接口的机器学习编程库 pycnml 将第2.1节的神经网络推断部分移植到 DLP 平台，实现手写数字分类。具体包括：

1) 利用提供 pycnml 库中的 Python 接口搭建手写数字分类的三层神经网络。

2) 熟悉在 DLP 上运行神经网络的流程，为在后续章节详细学习 DLP 高性能库以及智能编程语言打下基础。

3) 与第2.1节的实验进行比较，了解 DLP 相对于 CPU 的优势和劣势。

实验工作量：约需 2 个小时。

2.2.2 背景知识

2.2.2.1 量化

神经网络计算时通常采用 32 位的单精度浮点数（float32），而实际应用中 16 位定点数或 8 位定点数就可以满足精度需求。采用低位宽的定点数不仅可以有效减少存储空间及访存带宽，还可以有效减少运算器的面积和功耗，提高处理速度，例如 8 位定点乘法器的硬件开销约为 32 位浮点乘法器的 1/8。为了高能效地支持神经网络运算，DLP 只支持低位宽的定点数据类型，如 int8、int16。因此，在 DLP 上运行神经网络之前，需要对神经网络模型的参数（包括权重、偏置等）进行定点量化。

定点量化用一组共享指数位的定点数来表示一组浮点数，其中共享指数位表示二进制

小数的小数点的位置。常用的一种量化方式为：

$$q_x = round(\frac{r_x \times scale}{2^{position}}) \tag{2.22}$$

其中，r_x 表示输入的浮点数，q_x 表示定点量化后的整型数，$scale$ 是缩放因子，$position$ 是指数因子。

对神经网络模型量化时，首先需要运行一次 float32 数据类型的神经网络推断，获得每个网络层的输入数据及参数，然后对参数进行量化，获得该层的量化参数。为简化本节实验，本实验提供了量化后的网络模型，可以直接加载该模型参数进行实验。第4.2.2.3节将会详细介绍如何进行模型参数量化。

2.2.2.2 Python 接口的深度学习编程库 pycnml

深度学习编程库 pycnml，通过调用 DLP 上 CNML 库中的高性能算子实现了全连接层、卷积层、池化层、ReLU 激活层、Softmax 损失层等常用的网络层的基本功能，并提供了常用网络层的 Python 接口。pycnml 提供的编程接口可以用于在 DLP 上加速神经网络算法，具体接口说明如表2.2所示。pycnml 用 Python 封装了一个 C++ 类 CnmlNet，该类的成员函数定义了神经网络中层的创建、网络前向传播、参数加载等操作。

下面以图**??**为例，介绍如何调用 pycnml 提供的编程接口来创建网络层。首先实例化 pycnml.CnmlNet()，然后调用 CnmlNet 中的 createXXXLayer 成员函数就可以创建相应的网络层，例如创建全连接层时只需调用 pycml.CnmlNet().createMlpLayer。所有创建好的层对象的指针会按顺序以数组的形式保存在 CnmlNet 中，数组的下标作为层的 id 使用，当调用 pycnml.CnmlNet().loadParams 函数时，便可以通过此 id 来指定需要加载参数的层。pycml.CnmlNet().forward 函数会遍历层数组中的对象，依次调用每个层的前向传播函数，最终返回最后一层的前向传播结果。

```
# 实例化 CnmlNet
net = pycnml.CnmlNet()
# 设定网络输入维度
net.setInputShape(1, 3, 224, 224)
# conv1_1
# 创建卷积和全连接层时需要输入量化参数
net.createConvLayer('conv1_1', 64, 3, 1, 1, 1, input_quant_params[0])
# relu1_1
net.createReLuLayer('relu1_1')
```

图 2.14　pycnml 创建层程序示例

在使用 pycnml 之前，首先需要安装 pycnml 库：先解压 pycnml.tar.gz，再进入 pycnml 目录，执行 build_pycnml.sh 脚本进行编译和安装。安装完成后，进入 pycnml/env 目录下，执行 source env.sh 命令，之后便可以在 Python 程序中调用 pycnml 库。调用 DLP 的 pycml 库的 Python 程序，编译运行方式与 CPU 上的方式一致。

感兴趣的同学，可以进一步阅读附录A中的 C++ 程序示例，了解如何调用 CNML 库中的高性能算子实现全连接层的基本功能，ReLU 层和 Softmax 层的底层实现与之类似，具体每一层的 C++ 代码可以在 pycnml/src/layers 中查看。

表 2.2　pycnml 接口说明

接口	功能描述	参数/返回值
setInputShape： pycml.CnmlNet().setInputShape(dim_1, dim_2, dim_3, dim_4)	设定网络第一层输入数据的形状	dim_1 (int) : 维度 1 dim_2 (int) : 维度 2 dim_3 (int) : 维度 3 dim_4 (int) : 维度 4
createConvLayer: pycml.CnmlNet().createConvLayer(input_shape, out_channel, kernel_size, stride, dilation, pad, quant_param)	创建卷积层	input_shape (list) : 输入数据的形状，[N, input channel, input height, input width] output_channel (int) : 输出 channel 的大小 kernel_size (int) : 卷积核的大小 stride (int) : 卷积步长 dilation (int) : 膨胀系数 pad (int) : 填充大小 quant_param (QuantParam) : 量化参数
createMlpLayer: pycml.CnmlNet().createMlpLayer(input_shape, output_num, quant_param)	创建全连接层	input_shape (list) : 输入数据的形状，[N, input channel, input height, input width] output_num (int) : 输出数据的 channel 大小 quant_param (QuantParam) : 量化参数
createReLuLayer: pycml.CnmlNet().createReLuLayer(input_shape)	创建 ReLu 激活函数层	input_shape (list) : 输入数据的形状
createSoftmaxLayer: pycml.CnmlNet().createSoftmaxLayer(input_shape, axis)	创建 Softmax 损失层	input_shape (list) : 输入数据的形状 axis (int) : 进行 softmax 计算的维度
createPoolingLayer: pycml.CnmlNet().createPoolingLayer(input_shape, kernel_size, stride)	创建最大池化层	input_shape (list) : 输入数据的形状 kernel_size (int) : pool 窗口的大小 stride (int) : 窗口滑动步长
createFlattenLayer: pycml.CnmlNet().createFlattenLayer(input_shape, output_shape)	创建扁平化层	input_shape (list) : 输入数据的形状 output_shape (list) : 输出数据的形状
loadParams: pycnml.CnmlNet().loadParams(layer_id, filter_data, bias_data, quant_param)	为指定的层加载参数	layer_id (int) : 需要加载权重的层的 id。CnmlNet 中将创建的层存储在一个数组中，id 即为当前层在该数组中的下标，比如第一个层的 id 为 0，第二个层的 id 则为 1 filter_data (list) : 权重数据。必须是一维数组 bias_data(list) : 参数偏置 quant_param (QuantParam) : 量化参数
setInputData: pycml.CnmlNet().setInputData(input_data)	加载输入数据	input_data (list) : 输入数据。必须是一维数组，数据布局为 NCHW
forward: pycml.CnmlNet().forward()	进行前向传播计算	
getOutputData: pycnml.CnmlNet().getOutputData()	获取网络的计算结果	返回值 output_data : 网络最后一层的计算结果
size: pycnml.CnmlNet().size()	获取神经网络当前的层数	返回值 layers_num (int) : 当前层的数量
needToBeQuantized: pycnml.CnmlNet().needToBeQuantized(layer_id)	判断指定的层是否需要进行量化	返回值 need_to_be_quantized_or_not (bool)：当前层是否需要量化
QuantParam: pycnml.QuantParam	结构体，用于存放量化参数 position 和 scale。	该结构体可以通过构造函数来初始化，可以使用 pycnml.QuantParam(position:int, scale:float) 来创建一个 QuantParam 对象。 结构体成员： pycnml.QuantParam.position ： 获取当前 QuantParam 里存放的 position 参数。可以直接对其进行赋值。 pycnml.QuantParam.scale：获取当前 QuantParam 里存放的 scale 参数。可以直接对其进行赋值。

2.2.3 实验环境

硬件环境：DLP。

软件环境：pycnml 库、Python 编译环境及相关的扩展库，包括 Python 2.7.12，Pillow 6.0.0，Scipy 1.2.1，NumPy 1.16.0、CNML 高性能算子库、CNRT 运行时库

数据集：MNIST 手写数字库。

模型文件：量化参数文件、量化后的网络模型文件。

2.2.4 实验内容

使用 Python 封装的深度学习编程库 pycnml 搭建一个三层全连接神经网络，利用训练好的模型实现手写数字图像分类，并在 DLP 上正确运行。

与节类似，本实验的神经网络工程实现时大致分为以下 4 个模块：

1) 数据加载模块：读取测试数据并进行预处理。
2) 基本单元模块：不同网络层的定义，以及前向传播计算等基本功能。
3) 网络结构模块：利用基本单元模块搭建完整的网络。
4) 网络推断 (inference) 模块：使用已有的网络模型，对测试数据进行预测。

2.2.5 实验步骤

2.2.5.1 数据加载模块

本实验采用的数据集依然是 MNIST 手写数字库，数据读取的函数与第2.1.5.1 节的实现相同。因为本实验只需完成推断功能练，因此只用读取测试数据，进行预处理后存储在 Numpy 矩阵中，方便后续推断时快速从中读取数据，该部分代码如图2.15所示。

```
# file: mnist_mlp_demo.py
def load_data(self, data_path, label_path):
    # TODO: 调用函数load_mnist读取和预处理MNIST中训练数据和测试数据的图像和标记
    test_images = _____________
    test_labels = _____________
    self.test_data = np.append(test_images, test_labels, axis=1)
```

图 2.15 MNIST 子数据集的读取和预处理

2.2.5.2 基本单元模块

pycnml 库中已经将常用网络层的实现用 Python 语言封装起来，因此可以直接调用 pycnml 中的相关 Python 接口来实现神经神经网络的基本单元模块。具体调用方式，可以参照图2.14中的示例。

2.2.5.3 网络结构模块

网络结构模块可以直接使用 pycnml 封装好的基本单元接口来搭建一个完整的神经网络。在工程实现中，首先用一个类来定义一个神经网络，然后用类的成员函数来定义神经

网络的初始化、建立神经网络结构等基本操作。DLP 上实现的网络结构模块的程序示例如图2.16所示，定义了以下成员函数。

• 网络初始化：创建 pycnml.CnmlNet() 的实例 net，后续神经网络层的创建、参数的加载、前向传播计算等操作都通过该对象来调用。

• 建立网络结构：DLP 上只支持定点量化后的输入数据和权重，并且在创建全连接层时需要输入数据的量化参数。因此首先加载输入数据和权重的量化参数文件（量化参数包括指数因子 position 和缩放因子 scale），然后定义整个神经网络的拓扑结构。定义网络结构时，使用 net 中的 createXXXLayer 函数来实例化每一层。

```
# file: mnist_mlp_demo.py
class MNIST_MLP(object):
    def __init__(self):
        # 初始化网络，创建pycnml.CnmlNet() 实例
        self.net = pycnml.CnmlNet()
        self.input_quant_params = []  #输入数据的量化参数
        self.filter_quant_params = []   #模型参数的量化参数

    def build_model(self, batch_size=100, input_size=784,
                    hidden1=32, hidden2=16, out_classes=10,
                    quant_param_path='../data/mnist_mlp_data/mnist_mlp_quant_param.npz'):
        # 使用 pycnml 的接口建立三层神经网络结构
        self.batch_size = batch_size
        self.out_classes = out_classes

        # 读取量化参数
        params = np.load(quant_param_path)
        input_params = params['input']
        filter_params = params['filter']
        for i in range(0, len(input_params), 2):
            self.input_quant_params.append(pycnml.QuantParam(int(input_params[i]), float(
    input_params[i+1])))
        for i in range(0, len(filter_params), 2):
            self.filter_quant_params.append(pycnml.QuantParam(int(filter_params[i]), float(
    filter_params[i+1])))

        # 创建神经网络的层
        self.net.setInputShape(batch_size, input_size, 1, 1)
        # TODO: 使用 pycnml 搭建三层神经网络结构
        # fc1
        self.net.createMlpLayer('fc1', hidden1, self.input_quant_params[0])
        ____________________________________________
```

图 2.16　三层神经网络的网络结构模块 DLP 实现示例

2.2.5.4　网络推断模块

搭建好网络后，就可以加载训练好的模型，输入数据进行预测。网络推断模块的 DLP 实现程序示例如图2.17所示。神经网络推断模块的参数加载、前向传播、精度计算等基本操作拆分为神经网络类的成员函数来定义：

• 神经网络的参数加载：读取模型参数文件，并使用 net 中的 loadParams 接口加载参数。第2.1节实验中训练得到的模型参数用于本实验，但使用前需要使用量化工具对模型参

数（权重等）进行量化。为了便于使用，本实验提供了模型参数量化后的文件。将模型参数量化文件读入内存后，需要做两方面的处理：一方面，训练得到的模型中全连接层的权重的存放维度为 $C_{in} \times C_{out}$，而 DLP 处理全连接层时权重的处理维度为 $C_{out} \times C_{in}$，因此需要对读取的权重做一次转置；另一方面，由于 Python 中的浮点数类型 float 是双精度浮点，pycnml 接口内部实现的 C++ 函数接收的权重也只能是双精度浮点数类型，而 numpy 存储的数据包括权重都是 np.float32 类型，因此需要手动将 numpy 数据类型转为 np.float64 类型，否则在调用 pycml 库的接口过程中会报错。

- 神经网络的前向传播：net.forward 函数会自动遍历调用 net 中的每一层的前向传播函数，并将最后一层前向传播的计算结果返回。
- 神经网络推断函数主体：与第2.1节中的 CPU 实现类似，循环每次读取一定批量的测试数据，随后调用网络的前向传播函数计算得到神经网络的输出结果，然后与测试数据集的标记进行比对计算得到模型的精度。

```python
# file: mnist_mlp_demo.py
def load_model(self, param_dir):
    # TODO: 分别为三层全连接层加载参数
    params = np.load(param_dir).item()
    weigh1 = np.transpose(params['w1'], [1, 0]).flatten().astype(np.float)
    bias1 = params['b1'].flatten().astype(np.float)
    self.net.loadParams(0, weigh1, bias1, self.filter_quant_params[0] )
    weigh2 = np.transpose(params['w2'], [1, 0]).flatten().astype(np.float)
    bias2 = params['b2'].flatten().astype(np.float)
    ----------------------------
    weigh3 = np.transpose(params['w3'], [1, 0]).flatten().astype(np.float)
    bias3 = params['b3'].flatten().astype(np.float)
    ----------------------------

def forward(self):  # 前向传播
    return self.net.forward()

def evaluate(self):
    pred_results = np.zeros([self.test_data.shape[0]])
    # 读取一定批量的测试数据进行前向传播
    for idx in range(self.test_data.shape[0]/self.batch_size):
        batch_images = self.test_data[idx*self.batch_size:(idx+1)*self.batch_size, :-1]
        data = batch_images.flatten().tolist()
        # 加载输入数据
        self.net.setInputData(data)
        # 打印推理的时间
        start = time.time()
        self.forward()
        end = time.time()
        print('inferencing time: %f'%(end - start))
        prob = self.net.getOutputData()
        prob = np.array(prob).reshape((self.batch_size, self.out_classes))
        pred_labels = np.argmax(prob, axis=1)
        pred_results[idx*self.batch_size:(idx+1)*self.batch_size] = pred_labels
    accuracy = np.mean(pred_results == self.test_data[:,-1])
    print('Accuracy in test set: %f' % accuracy)
```

图 2.17　三层神经网络的网络推断模块 DLP 实现示例

2.2.5.5 完整实验流程

完成所有模块的实现后，就可以调用上述模块中的函数，在 DLP 上运行神经网络实现手写数字图像分类。网络运行的流程与 CPU 上的执行流程基本一致。本实验中三层神经网络的完整流程的程序示例如图2.18所示。首先实例化三层神经网络对应的类；其次调用网络结构模块 build_model 建立神经网络，指定神经网络的超参数（如每层的神经元个数）；随后调用 load_data 函数进行数据的加载和预处理；然后调用 load_model 函数从文件中读取训练好的模型参数；最后调用 evaluate 函数执行网络推断模块获得预测结果，并测试网络精度。上一节的 CPU 实验中，我们设置的默认 batch size 为 100，对于这种小规模的运算，使用 DLP 这样算力强大的设备实在是有些大材小用了，因此我们将 batch size 改为 10000，一次传入 10000 张图片，记录 DLP 计算的时间。因为 CNML 在第一次运行的时候会有一个指令生成的过程，导致运行时间会长一些，所以我们多次执行 evaluate 函数，排除第一次计算的时间，其它的每次计算时间就和真实的硬件时间很接近了。

```
# file: mnist_mlp_demo.py
if __name__ == '__main__':
    # 设置 batch_size 为 10000
    batch_size = 10000
    h1, h2, c = 32, 16, 10
    mlp = MNIST_MLP()
    mlp.build_model(batch_size=batch_size, hidden1=h1, hidden2=h2, out_classes=c)
    model_path = '../data/mnist_mlp_data/mlp-%d-%d-10epoch.npy'%(h1,h2)
    test_data = '../data/mnist_mlp_data/mnist_data/t10k-images-idx3-ubyte'
    test_label = '../data/mnist_mlp_data/mnist_data/t10k-labels-idx1-ubyte'
    mlp.load_data(test_data, test_label)
    mlp.load_model(model_path)
    # 循环多次统计 DLP 计算时间
    for i in range(3):
        mlp.evaluate()
```

图 2.18　三层神经网络的完整流程 DLP 实现示例

2.2.6 实验考核

本实验中，精度评判标准与第2.1节实验一样，使用测试集的平均分类正确率判断分类结果的精度。性能评判标准为设置 batch size 为 10000 时，进行一次 forward 的时间。本实验的评分标准设定如下：

- 60 分标准：完善本节实验代码，用 pycnml 搭建出的三层神经网络能够在 DLP 上进行推断，并且在测试集上的平均分类正确率高于 90%。
- 80 分标准：修改网络隐藏层的数量，使用第 2.1 实验的代码重新训练模型，使训练得到的模型在 DLP 上运行的推断（forward）耗时为 CPU 推断耗时的 1/20 或更低，并且在测试集上的平均分类正确率高于 95%。
- 100 分标准：修改网络隐藏层的数量，使用第 2.1 实验的代码重新训练模型，使训练得到的模型在 DLP 上运行的推断耗时为 CPU 推断耗时的 1/50 或更低，并且在测试集上的平均分类正确率高于 98%。

2.2.7 实验思考

在实验中请思考如下问题：

1) DLP 在进行神经网络推断时相对于 CPU 有什么优势和劣势?

2) 在什么样的神经网络结构下，DLP 能够最大发挥它的性能优势?