第 1 章　绪论

智能计算系统是智能时代的核心物质载体。要理解智能计算系统软硬件技术栈，需要融会贯通地掌握智能算法、编程框架、智能编程语言、芯片架构等四个方面的知识。读者要真正理解智能计算系统、掌握相应的能力，在学习了《智能计算系统》教材中的理论知识之后，还必须通过实际动手编程点亮智能计算系统知识树。这就是本实践教程的目标。

延续《智能计算系统》教材，本书仍以风格迁移作为驱动范例，针对智能算法、编程框架、智能编程语言、芯片架构中的知识点设计了相应的实验及拓展思考（分阶段实验）。读者通过实验点亮相应的知识点，进而将软硬件技术栈贯通起来，最终就能开发出一个实际的完成图像风格迁移任务的智能计算系统。在此基础上，读者还可以开发出来更多其他的智能计算系统（综合实验），巩固系统能力。

本章首先简单介绍智能计算系统的概念及发展，然后介绍本书的实验设计，最后介绍本书的实验平台。

1.1　智能计算系统简介

人工智能是人制造出来的机器所表现出来的智能。人工智能通常分为两大类：弱人工智能和强人工智能。弱人工智能是能够完成某种特定具体任务的人工智能，而强人工智能是具备与人类同等智慧或超越人类的人工智能。目前广泛应用于图像识别、语音识别、自然语言处理、博弈游戏等应用上的深度学习技术，就属于弱人工智能。本实践教程重点关注面向弱人工智能的智能计算系统。

一个完整的智能体需要能够从外界获取输入，并将智能处理结果输出给外界。而人工智能算法或代码本身并不能构成一个完整的智能体，必须要在一个具体的物质载体上运行起来才能作为一个完整的智能体作用于物质世界，展现出智能。因此，智能计算系统是人工智能的物质载体。

传统以通用 CPU 为中心的计算系统的速度和能效难以满足智能应用的需求，因此现阶段智能计算系统通常是集成通用 CPU 和深度学习处理器（DLP，Deep Learning Processor）的硬件异构系统，同时包括一套面向开发者的智能计算编程环境（包括编程框架和编程语言），该编程环境可以方便程序员快速便捷地开发高能效的智能应用程序。因此，智能计算系统覆盖深度学习算法、编程框架、智能编程语言、深度学习处理器等软硬件技术栈，如图1.1所示。

深度学习（多层大规模神经网络）算法是当前智能计算系统的核心人工智能算法。自 2012 年深度卷积神经网络（Convolutional Neural Network，CNN）——AlexNet 获得 ImageNet 大规模视觉识别比赛冠军，深度学习得到业界广泛关注。随着数据集和模型规模的快速发展，深度学习的识别精度越来越高，已经广泛用于语音识别、人脸识别、机器翻译等领域，甚至在围棋和《星际争霸》等游戏中战胜了人类顶级高手，并形成了图像风格迁移等有意思的应用。面向不同应用领域，已经演化出并不断迭代出不同种类的新的深度学习算法，例

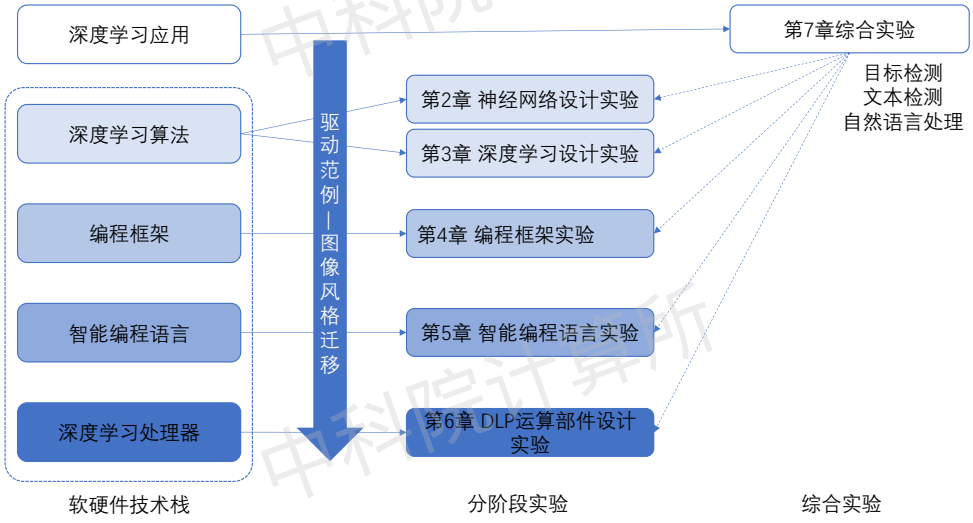

图 1.1　智能计算系统的软硬件技术栈及实验内容

如用于图像分类的 CNN 网络（如 VGG、ResNet 等）、用于图像目标检测的 CNN 网络（如 R-CNN 系列、YOLO 等）、用于序列信息处理的循环神经网络（Recurrent Neural Network，RNN）及长短期记忆网络（Long Short-Term Memory，LSTM）、生成对抗网络（Generative Adversarial Net，GAN）等。

将种类繁多且快速迭代的深度学习算法高效地在多种智能芯片上运行起来，需要编程框架的支持。编程框架是智能计算系统中非常关键的核心枢纽，发挥承上启下的作用。对于程序员，编程框架将智能算法中的常用操作（如卷积、池化等）封装成算子可以被直接调用，降低智能应用的开发难度，提高应用的开发效率；对于智能芯片，编程框架将智能算法拆分出的一系列具体算子分配到智能芯片（或 CPU）上运行，以达到更优的运行性能。2014 年加州大学伯克利分校发布的深度学习编程框架 Caffe，是出现最早的框架之一，由于其易用、稳健、高效的特点，被广泛用于深度学习的训练和预测。2015 年底谷歌发布的编程框架 TensorFlow，支持自动求导、训练好的模型可以部署到不同的硬件/操作系统平台上，是目前最受欢迎的框架之一。此后，出现了 MXNet、PyTorch、PaddlePaddle 等编程框架。今天编程框架的易用性极大地推动了深度学习算法的发展。

智能编程语言是智能计算系统中连接智能编程框架和智能计算硬件的桥梁。由于深度学习处理器架构与传统通用 CPU 在控制、存储及计算等逻辑上都有较大区别，传统编程语言（如 C/C++、Java、Python、汇编语言等）在面向智能计算系统编程时难以同时满足高开发效率、高性能和高可移植性的需求，因此需要有新的高级智能编程语言。为适应人工智能算法和深度学习处理器的快速演进，智能编程语言需要对不同规模、不同尺度及不同形态的智能计算系统进行层次化的硬件架构抽象，并在此基础上为用户提供简洁统一的编程接口。例如，中科院计算所提出了智能编程语言 BANG C Language（BCL），不仅可以提升智能算法的开发效率，还可以利用深度学习处理器的结构特点来有效应对不断演进的深度学习算法。此外，面向图像处理的 Halide、面向深度学习的 RELAY/TVM 等领域专用语言，也对特定领域的应用和硬件进行了一定程序的抽象。

深度学习处理器是智能计算系统的核心，近年来得到了快速发展。2013 年，中科院计算所和法国的 Inria 共同设计了国际上首个深度学习处理器架构——DianNao，可以灵活、高效地处理上百层、千万神经元、上亿突触的各种深度学习神经网络（甚至更大）；且相对传统通用 CPU，可以取得两个数量级（甚至更高）的能效优势。随后中科院计算所和法国 Inria 又设计了国际上首个多核深度学习处理器架构 DaDianNao 和首个机器学习处理器架构 PuDianNao。进一步，中科院计算所提出了国际上首个深度学习指令集 Cambricon。中科院计算所还研制了国际上首款深度学习处理器芯片“寒武纪 1 号”。目前寒武纪系列处理器已实用近亿台智能手机和服务器中，推动了深度学习处理器从理论走向实际，普惠大众。此外，近年来，Google、NVIDIA、Intel、IBM、MIT、Stanford 等公司和研究机构都在引用计算所的 DianNao 系列论文，开展深度学习处理器方面的研制工作。

得益于深度学习算法、编程框架、智能编程语言、深度学习处理器等方面的技术进步，智能计算系统已成为计算机的一类主流形态。今天，大量的超级计算机、数据中心计算机、智能手机、智能物端等都是以深度学习类应用为核心负载，因此都在朝智能计算系统方向演进。例如，IBM 将其研制的 2018 年世界上最快的超级计算机 SUMMIT 称为智能超算。在 SUMMIT 上利用深度学习方法做天气分析的工作甚至获得了 2018 年超算应用最高奖——Gorden Bell 奖。数据中心计算机利用深度学习做广告推荐、自动翻译、智能在线教育、智慧医疗等应用，是典型的智能计算系统。手机更是因其要用深度学习处理大量图像识别、语音识别、自动翻译等任务，被广泛看作一种典型的小型智能计算系统。仅集成寒武纪深度学习处理器的手机就已有近亿台。智能物端包括机器人、自动驾驶、手表、监控等也广泛使用深度学习进行相关任务的处理。因此，未来如果人类社会真的进入智能时代，可能绝大部分计算机都是智能计算系统。

本实践教程主要面向深度学习的智能计算系统。

1.2 实验设计

结合智能计算系统的软硬件技术栈，本书设计了如图1.1所示的分阶段实验和综合实验。其中，分阶段实验以图像风格迁移作为驱动范例，通过逐步完成算法实验（第 2-3 章）、编程框架实验（第 4 章）、智能编程语言实验（第 5 章）、DLP 运算部件实验（第 6 章）等，点亮知识树（如图1.2所示），开发出实现图像风格迁移的智能计算系统；综合实验包括目标检测、文本检测、自然语言处理等不同应用领域的实验，巩固对相关知识的系统理解和掌握，了解不同应用对智能计算系统的需求，并开发出相应的智能计算系统，进阶为智能计算系统全栈工程师。

针对上述实验，我们首先介绍实验目标和相关背景知识；之后介绍相关实验环境（为了由浅入深地增进读者对智能计算系统地了解，实验环境包括通用 CPU 平台和深度学习处理器平台）；接下来通过详细实验步骤引导读者进行实验并给出评估标准；最后，在实验思考部分引导读者进行进阶设计，以加深对智能计算系统的全面理解。

第 2 章介绍如何利用三层全连接神经网络实现手写数字分类的两个实验，包括在 CPU 平台和 DLP 平台上实现手写数字分类，以帮助读者深入理解神经网络训练及预测原理。其中，实验一采用 Python 语言实现神经网络的基本单元，包括全连接层、激活函数、损失函

图 1.2　智能计算系统知识树

数（涉及知识可参考《智能计算系统》教材第 2.3 节），该实验有助于理解神经网络中正向传播和基于随机梯度下降法的反向传播的原理及计算复杂度（涉及知识可参考《智能计算系统》教材第 2.2 节）。此外，通过训练一个简单完整的神经网络，以帮助读者理解隐层个数、神经元数、激活函数、损失函数和学习率等对网络训练及分类精度的影响。通过对网络层的作用及层间关系的深入理解，为后续更复杂的综合实验（如风格迁移等）奠定基础。实验二调用 DLP 上 Python 语言封装的深度学习编程库 pycnml，将实验一中神经网络前向相关的模块移植到 DLP 平台上，最终在 DLP 平台上实现手写数字分类。通过该实验，读者可以对 DLP 编程和处理效率有初步了解。

> 知识点：（图1.2中黄色部分）全连接神经网络的正向传播、随机梯度下降法、反向传播，以及设计优化方法包括网络结构（隐层个数、神经元个数）、激活函数、损失函数等。

第 3 章介绍深度学习算法相关的三个实验，包括 CPU 平台、DLP 平台上的 VGG 图像分类以及非实时风格迁移等。其中，VGG 是风格迁移应用中用到的深度学习算法，涉及到卷积层、池化层、全连接层、softmax 损失函数等（涉及知识可参考《智能计算系统》教材第 3.1 节）。其中，实验一是 VGG 图像分类实验，主要面向如何用 Python 实现 VGG 网络结构、加载模型参数、模型分类测试，并分析 VGG 网络的计算量及性能瓶颈，为风格迁移实验中使用 VGG 网络计算风格特征相似度做准备。实验二是 DLP 平台的图像分类实验，需要调用 pycnml 库中的相关接口将实验一中的相关模块移植到 DLP 平台上，最终在 DLP 平台上实现图像分类。实验三是非实时风格迁移实验（对应《智能计算系统》教材第 3.6.1 节），利用 VGG 对输入图像进行训练来获得风格化后的图像，包括如何用 VGG 网络提取图像特征，如何计算内容/风格损失，如何迭代训练来求解风格化图像等。

知识点：（图1.2中棕色部分）卷积神经网络的网络层（包括卷积层、池化层、全连接层、softmax 层等）及构建，基于 VGG 的图像分类，基于 VGG 的非实时风格迁移算法。

第 4 章介绍深度学习编程框架相关的四个实验，包括利用编程框架实现图像分类、利用图像转换网络预测实现实时风格迁移、图像转换网络的训练、编程框架的 CPU 自定义算子。其中，实验一是图像分类实验，介绍如何利用 TensorFlow 框架实现 CPU 和 DLP 两种平台上的图像分类，帮助读者熟悉 TensorFlow 的编程模型及基本用法。实验二是实时风格迁移预测实验，在实验一的基础上基于 TensorFlow 实现实时风格迁移中图像转换网络的预测。其中涉及多种网络层的定义、神经网络模型的创建、模型参数的加载、神经网络模型的计算及输出等。实验三是实时风格迁移训练实验，基于 TensorFlow 实现图像转换网络的训练，包括加载数据并进行预处理、对模型迭代训练、实时保存模型结果等。通过与第 3 章使用 Python 语言实现深度学习算法对比，读者可以体会采用编程框架开发的便利性和高效性。实验四是 CPU 自定义算子实验，介绍如何在 TensorFlow 中新增自定义算子，以解决原生编程框架不支持特定算子的问题。

知识点：（图1.2中绿色部分）编程框架 TensorFlow 的编程模型的基本用法，基于 TensorFlow 的图像分类以及实时图像风格迁移预测（即 VGG 网络的前向传播，包括定义模型计算单元（如卷积层、池化层）、创建网络模型），基于 TensorFlow 的实时深度风格迁移训练（即模型训练，包括定义损失函数、创建优化器、定义模型训练方法、保存模型），自定义 TensorFlow 算子。

第 5 章介绍智能编程语言方面的两个实验，包括 BCL 算子开发与集成、BCL 性能优化。其中，实验一是 BCL 算法开发与集成实验，介绍如何使用智能编程语言 BCL 定义新的算子以扩展高性能库算子，并将其集成到编程框架中，从而加速实时图像风格迁移。通过该实验，可以掌握对高性能库及编程框架进行扩展的能力，并可以根据特定应用场景需求自主定义 DLP 算子，以满足快速演进的智能算法的需求。实验二是 BCL 性能优化实验，以矩阵乘为例，介绍如何利用智能编程语言 BCL 来充分利用 DLP 上的计算和存储资源实现性能优化。通过该实验，掌握 DLP 平台算法性能瓶颈分析方法、多核流水优化技术，从而加深对智能计算系统和智能编程语言的理解和应用。

知识点：（图1.2中橙色部分）智能计算系统抽象架构（包括计算模型、控制模型、存储模型），智能编程模型（包括 kernel 函数、编译器支持和运行时支持），智能编程语言基础，面向智能计算设备的高层接口——智能应用编程接口，功能调试接口及工具，性能调优接口及工具。

第 6 章* 以深度学习算法中时间最核心的卷积运算和矩阵运算为例，介绍如何设计深度学习处理器运算器，包括串行内积运算器、并行内积运算器、以及矩阵运算子单元等。其中，串行、并行内积运算器实验，分别介绍如何使用 Verilog HDL（Hardware Description Language，硬件描述语言）编写实现深度学习卷积和全连接层中的内积运算，然后在 Modelsim 仿真环境下进行仿真。为保证该内积运算器是真正通过具体物理电路实现的，不仅其内部各模块是可仿真且可综合成门级网表，还需要评估内积运算器的性能。在前两个实验基础上，矩阵内积运算子单元实验，介绍如何设计运算单元的整体架构，如何设计各子模块的功能、接口和结构，实现具体的 Verilog 代码，并通过仿真评估矩阵运算单元的性能。第 6 章实验供有芯片设计基础的同学选做。

知识点：（图1.2中蓝色部分）算法计算特征和访存特征分析、深度学习运算器设计、编译调试仿真。

第 7 章综合实验中介绍了目标检测、文本识别和自然语言处理三个不同领域的人工智能应用。通过将智能算法、编程框架、智能编程语言和深度学习处理器的相关知识点串联起来，在智能计算平台上实现应用部署及优化，从而使读者具备融会贯通的智能计算系统设计开发能力。其中，实验一是目标检测实验，介绍面向 DLP 平台如何实现经典的目标检测算法——YOLOv3 网络，并进行性能优化和离线部署，最终完成在 DLP 平台上的目标检测应用。实验二是文本检测实验，介绍面向 DLP 平台如何实现文本检测的代表性算法——EAST 网络，并进行性能优化和离线部署，最终完成在 DLP 平台上的目标检测应用。实验三是自然语言处理实验，介绍如何实现自然语言处理的代表性算法——BERT 网络，并进行性能优化和离线部署，最终完成在 DLP 平台上的自然语言处理应用。

知识点：第 2-6 章实验相关知识点，以及（图1.2中深蓝色部分）目标检测（基于 YOLO 网络），文本检测（基于 EAST 网络的），自然语言处理（基于 BERT 网络）。

标* 部分属于拓展内容，供读者选做。

1.3 实验平台

为配合相关实验的开展，我们采用的实验平台包括通用 CPU 平台和智能计算系统平台。其中智能计算系统平台集成了深度学习处理器硬件，并提供了配套软件开发环境。

1.3.1 硬件平台

本书实验所采用的 DLP 芯片内部集成了 4 个深度学习处理器簇（Cluster），其中每个 Cluster 包括 4 个智能处理器核及 1 个存储核。每个智能处理器核包括并行的向量和矩阵运算单元、神经元存储单元（Neuron RAM，NRAM）和权值存储单元（Weight RAM，WRAM），

而存储核中则包括共享的片上存储（Shared RAM，SRAM）。其具体结构如图 6.1所示。该 DLP 硬件以 PCIe 加速卡的形式提供给用户使用，其峰值算力为 128T，支持包括 INT16、INT8、INT4、FP32 及 FP16 等多种不同的数据类型，满足多样化的智能处理需要，兼具通用性和高性能。

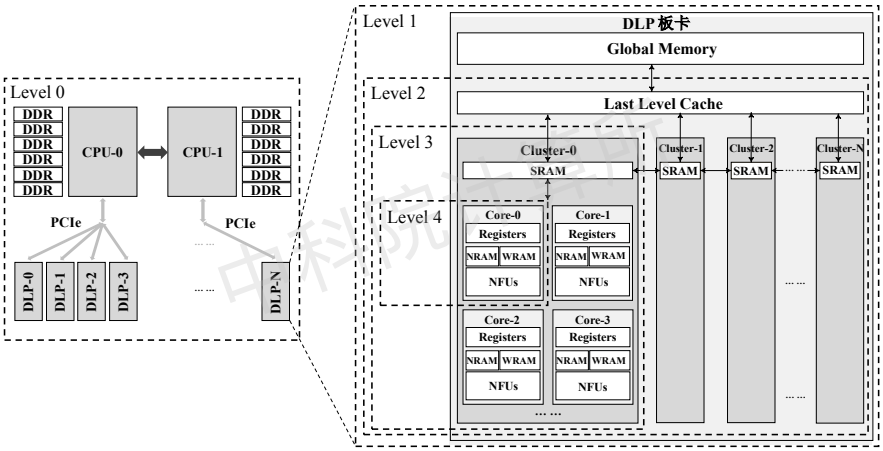

图 1.3　实验所采用的 DLP 硬件架构

除了直接在 PC 或者服务器中使用 DLP 硬件进行编程外，我们同时提供了云平台环境方便用户使用 DLP 硬件。云平台的基本功能和使用方法参见附录章节 B 或课程网址 http://novel.ict.ac.cn/aics/。

1.3.2　软件环境

DLP 硬件的整体软件环境如图 1.4所示，大致包括 6 个部分：编程框架、高性能库 CNML、智能编程语言 BCL 及编译器、运行时库 CNRT 及驱动、开发工具包及领域专用开发包等。其中，编程框架包括 TensorFlow、PyTorch 和 Caffe 等。DLP 上的高性能库 CNML 提供了一套高效、通用、可扩展的编程接口，用于在 DLP 上加速各种智能算法。用户可以直接调用 CNML 中大量已优化好的算子接口来实现其应用，也可以根据需求扩展算子。智能编程语言 BCL 可以用于实现编程框架和高性能库 CNML 中的算子。DLP 的运行时库 CNRT 提供了面向设备的用户接口，用于完成设备管理、内存管理、任务管理等功能。运行时库作为 DLP 软件环境的底层支撑，其他应用层软件的运行都需要调用 CNRT 接口。除了上述基本软件模块外，还提供了多种工具方便用户进行状态监测及性能调优，如应用级性能剖析工具、系统级性能监控工具和调试器等。上述具体内容将在后续实验的背景部分介绍。

上层智能应用可以通过两种方式来运行：在线方式和离线方式。其中，在线方式直接用各种编程框架（如 TensorFlow、PyTorch、MXNet 和 Caffe 等）间接调用高性能库 CNML 及运行时库 CNRT 来运行。离线方式通过直接调用运行时库 CNRT，运行前述过程生成的特定格式网络模型，减少软件环境的中间开销，提升运行效率。

此外，为配合相关实验开展，我们还提供了配套的自动评分实验教学系统。该系统可以自动对读者完成的各项实验进行评分。

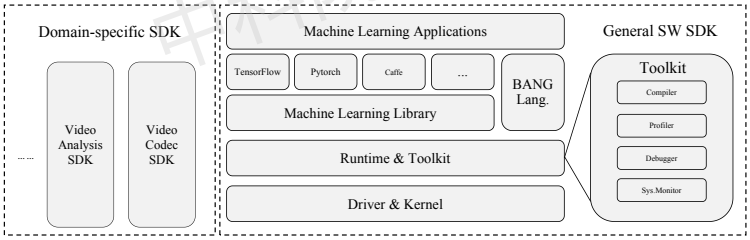

图 1.4　实验所采用的 DLP 硬件的软件环境

1.4　本章小结

本章首先简单介绍了智能计算系统的概念。为了帮助读者拥有实际开发智能计算系统的能力，本书仍以风格迁移为例，介绍了深度学习算法、编程框架、智能处理器和智能编程语言等方面的分阶段实验，逐步实现风格迁移的智能计算系统开发。最后，以不同应用领域的综合实验巩固对智能计算系统软硬件栈的相关知识的系统理解和掌握。

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。首先介绍在 CPU 平台上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，随后介绍如何将前述算法移植到深度学习处理器 DLP 上。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的推断功能。完成本章实验，读者就可以点亮智能计算系统知识树（图1.2）中神经网络算法部分的知识点。

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推断方法，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（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 实现神经网络训练所使用的梯度下降算法，加深对神经网络训练过程的理解。

实验工作量：约 20 行代码，约需 2 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；输出为一维向量 $\boldsymbol{y}$，维度为 n；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。前向传播时，全连接层的输出的计

[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。

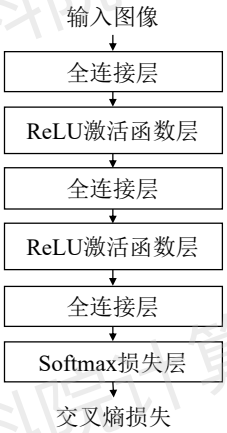

图 2.1　用于手写数字分类的三层全连接神经网络

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 $\nabla_{\boldsymbol{x}}L = \frac{\partial L}{\partial \boldsymbol{x}}$ 计算公式分别为：

$$\begin{aligned}
\nabla_{\boldsymbol{W}}L &= \boldsymbol{x}\nabla_{\boldsymbol{y}}L^T \quad \text{L对y的偏导然后转置} \\
\nabla_{\boldsymbol{b}}L &= \nabla_{\boldsymbol{y}}L \\
\nabla_{\boldsymbol{x}}L &= \boldsymbol{W}^T\nabla_{\boldsymbol{y}}L \quad \text{此处w不需要加转置}
\end{aligned} \tag{2.2}$$

实际应用中通常使用批量随机梯度下降算法进行反向传播计算，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

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

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 $\boldsymbol{1}$ 相乘。

ReLU 激活函数层

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，对 $\boldsymbol{x}$ 计算 e 指数并进行归一化，即得到 Softmax 分类概率。假设 $\boldsymbol{x}$ 对应 i 位置的值为 $\boldsymbol{x}(i)$，$\hat{\boldsymbol{y}}(i)$ 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层的输出计算本层输入的损失。对于 Softmax 损失层，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，则对每个输入计算 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。

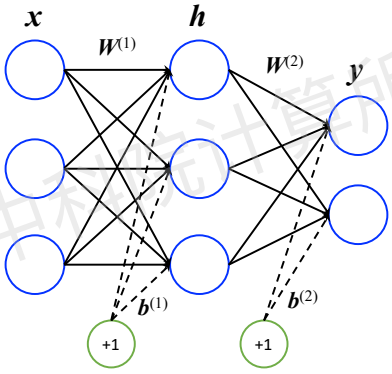

图 2.2　两层神经网络示例

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 损失层组成[①]，其中第一个全连接层的权重为 $\boldsymbol{W}^{(1)}$，偏置为 $\boldsymbol{b}^{(1)}$，

[①]本实验中实现的三层神经网络与这个例子非常类似，即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

第二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 损失层的损失值 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。

2.1.2.3 精度评估

在图像分类任务中，通常使用测试集的平均分类正确率来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 $N = 10000$），$\boldsymbol{p}_i$ 为神经网络输出的第 i 张图像的推断结果，$\boldsymbol{p}_i$ 为一个向量，取其中最大分量对应的类别作为推断类别。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

$$R = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}(\text{argmax}(\boldsymbol{p}_i) = y_i) \tag{2.19}$$

其中 $\mathbf{1}(\text{argmax}(p_i) = y_i)$ 表示当 $\boldsymbol{p}_i$ 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。

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 2.7.12，Pillow 6.0.0，Scipy 0.19.0，NumPy 1.16.0（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的工程实现通常划分为 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。

本实验即采用上述较为细致的模块划分方式。需要说明的是，目前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。在某些特殊的应用场景下，神经网络工程还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），非实时风格迁移工程中不包含推断模块（如实验3.3）。

①在不同文献中可能被称为测试、推断或预测。

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。

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

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。

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，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 激活函数层和 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

```python
# file: mnist_mlp_cpu.py
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
# file: mnist_mlp_cpu.py
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

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存到指定的文件中。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。

以下是全连接层、ReLU 激活函数层和 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，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：

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

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \eta \nabla_{\boldsymbol{b}} L \tag{2.21}$$

• 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 激活函数层：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：

• 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
# file: layers_1.py
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: 利用梯度对全连接层参数进行更新
        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

```python
# file: layers_1.py
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 激活函数层的实现示例

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。

```python
# file: layers_1.py
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。

```python
# file: mnist_mlp_cpu.py
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

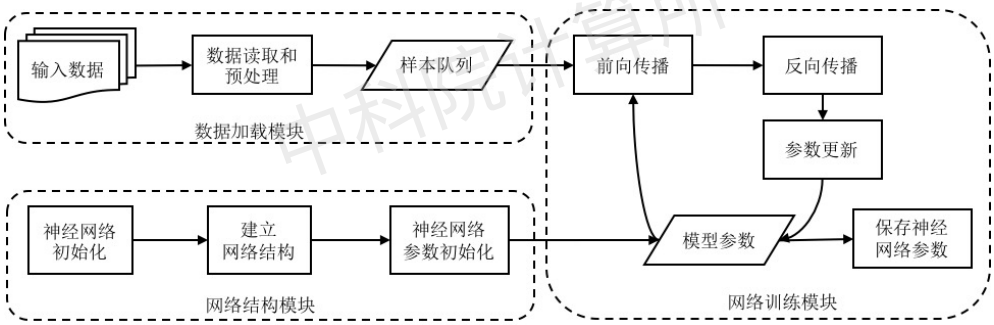

图 2.9　神经网络训练流程

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结构，顺序调用每层的前向传播函数。以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结构，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，因此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

```python
# file: mnist_mlp_cpu.py
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

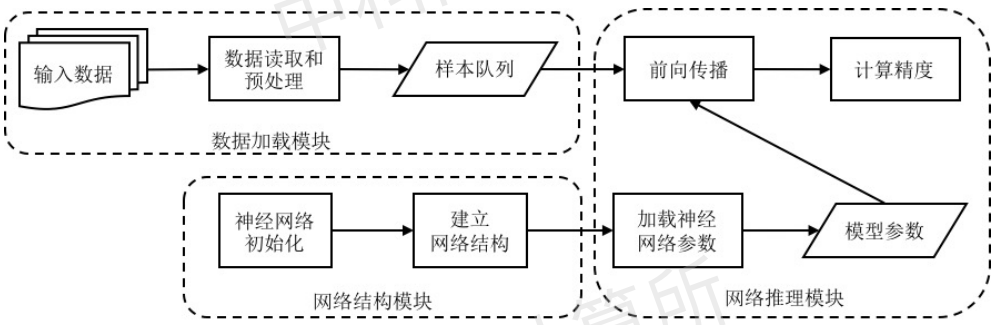

图 2.11　神经网络推断流程

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，再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。

```
# file: mnist_mlp_cpu.py
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

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。

```
# file: mnist_mlp_cpu.py
def build_mnist_mlp(param_dir='weight.npy'):
    h1, h2, e = 32, 16, 10
    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))
    return mlp

if __name__ == '__main__':
    mlp = build_mnist_mlp()
    mlp.evaluate()
```

图 2.13　三层神经网络的完整流程实现示例

2.1.5.7 实验运行

根据第2.1.5.1节 ~ 第2.1.5.6节的描述补全 layer_1.py、mnist_mlu_cpu.py 代码，并通过 Python 运行.py 代码。具体可以参考以下步骤。

1. 环境申请

按照附录B说明申请实验环境并登录云平台，本实验的代码存放在云平台/opt/code_chap_2_3/code_cl目录下。

```
# 登录云平台
ssh root@xxx.xxx.xxx.xxx -p xxxxx
# 进入 code_chap_2_3_student 目录
cd /opt/code_chap_2_3/code_chap_2_3_student
# 初始化环境
source env.sh

```

2. 代码实现

补全 stu_upload 中的 evaluate_cpu.py、evaluate_mlu.py 文件。

```
# 进入实验目录
cd exp_2_1_mnist_mlp
# 补全 layers_1.py, mnist_mlp_cpu.py
vim stu_upload/layers_1.py
vim stu_upload/mnist_mlp_cpu.py

```

3. 运行实验

```
# 运行完整实验
python main_exp_2_1.py

```

2.1.6 实验评估

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

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 10 行代码，约需 1 个小时。

2.2.2 背景知识

2.2.2.1 量化

在通用处理器上实现时，神经网络的权重、偏置、激活值等信息通常会用浮点数 float32 来表示。当神经网络规模较大时，网络参数随之增多，对深度学习处理器的计算能力、存储空间及访存带宽都会带来巨大的压力。工作[3] 表明，对权重、偏置等参数用低精度的数据表示，即模型量化，对神经网络的精度不会产生很大的影响，同时可以显著加速神经网络的推理和训练速度。

深度学习处理器 DLP 上的模型量化通常采用定点量化，即将浮点数映射到定点数来表示，如图2.14所示，通常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 position 确定了二进制小数的小数点位置。通过定点量化可以大幅降低数据的存储空间，例如，将 float32 量化成 int8 后，存储空间可以减少为原来的 1/4。为了高能效地支持神经网络运算，DLP 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。

目前 DLP 上支持在线量化和离线量化两种方式。其中，离线量化是 DLP 上最常用的量化方式。离线量化时，用户通过调用相应的接口、设置表2.2中量化参数值（*positionscale*）就可以完成量化。

DLP 上的离线量化有两种，包括对称定点表示和有缩放系数的对称定点表示

• 对称定点表示：对实数数据 r_x 直接用整型数据 q_x 左移 *position* 位。其量化和反量化过程如下

$$q_x = round(\frac{r_x}{2^{position}})$$

表 2.2　量化数据及参数

数学符号	含义
r_x	需要定点量化的实数
q_x	定点量化后的整数
$position$	小数点的位置
$scale$	缩放系数
$offset$	偏移量
$round$	四舍五入取整
n	量化位宽（例如 int8，n=8）

$$r_x \approx q_x \times 2^{position}$$

• 有缩放系数的对称定点表示：先对实数数据做缩放，然后做对称定点表示处理。其量化和反量化过程如下

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

$$r_x \times scale \approx q_x \times 2^{position}$$

通过公式可以发现有缩放系数的对称定点表示是对称定点表示的改进版，如图 2.14所示，设 Z 为需要量化表示的数域中所有数的绝对值最大值 $max(r_x)$，则 A 需要包含 Z，且 Z 要大于 $A/2$；$position$ 的计算和对称定点表示相同，其中 n 表示量化类型的位宽，例如 int8 的位宽为 8，int16 的位宽为 16。$position$ 计算表示如下：

$$position = ceil(log_2(\frac{max(r_x)}{2^{n-1}-1}))$$

$$A = 2^{ceil(log_2(\frac{max(r_x)}{2^{n-1}-1}))}(2^{n-1}-1)$$

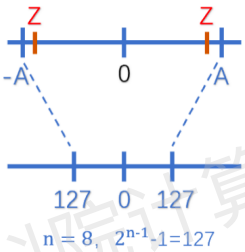

图 2.14　int8 量化

同时，$position$ 取值需要满足: 如果 $position$ 减小 1，就不能够覆盖需要量化的实数集合的最大值 $max(r_x)$，即:

$$(2^{n-1}-1) \times 2^{position-1} < max(r_x)$$

由此在求得 position 之后就可以比较简单的求得 scale 的值。

$$scale = \frac{max(r_x)}{(2^{n-1}-1) \times 2^{position}}$$

目前 DLP200 系列支持量化到 int4、int8、int16 三种类型。量化后数据位宽越低，则吞吐量越大，计算越快，对结果的精度影响也会增大。建议只用在带 filter（例如 conv、mlp 等复杂算子）的计算过程中，对于简单算子 (如 add)，设置之后对速度提升不大，反而会降低精度。

量化中常见的两种处理方法：

1. 对普通权重量化，即整个权重或输入内的所有数据都用相同的量化参数。

2. 按通道对权重或者输入进行量化，即整个张量内的数据按照不同的通道，每个通道内的数据用相同的参数，不同通道使用的参数可能不同。

本实验使用对普通权重量化的方式，为简化实验流程，此处提供了针对所有数据的量化参数，可以直接使用该参数进行实验。理论上修改了网络结构重新训练新的模型是需要重新计算量化参数的，但是经过测试，在本实验中如果仅修改网络隐藏层数量，采用旧的量化参数对结果精度的影响非常的小，因此本实验中不需要重新进行量化参数的计算。

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2.3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2.15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 2.15　pycnml 创建层程序示例

在使用 pycnml 之前，首先需要安装 pycnml 库：先执行 source env.sh 命令初始化环境，然后解压 pycnml.tar.gz，再进入 pycnml 目录，执行 build_pycnml.sh 脚本进行编译和安装。安装完成后便可以在 Python 程序中调用 pycnml 库，编译运行方式与 CPU 上的方式一致。如果实验中采用的是云平台环境，由于云平台中已经集成了 pycnml 库，则不需要再手动安

表 2.3　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
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。

装 pycnml。

感兴趣的同学，可以进一步阅读 pycnml 源码中层的实现，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。

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 0.19.0，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2.1.4节类似，本实验的神经网络工程实现大致分为以下 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，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读取数据，该部分代码如图2.16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 2.16 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 接口来实现神经网络的基本单元模块。具体调用方式可以参照图2.15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以下代码所示：

```
# 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])
        ____________________________________________
```

上述示例程序中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。

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 实现程序示例如以下代码所示：

```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)
```

上述示例程序中，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。可以使用第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（如权重等）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。

2.2.5.5 完整实验流程

完成所有模块的实现后，就可以在 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。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。其中，用 load_data 和 load_model 加载数据和参数时，首先会在 DLP 上分配内存空间，然后将数据或参数从主存拷贝到 DLP 的存储。本实验完整流程的程序示例如下代码所示：

```
# file: mnist_mlp_demo.py
# 神经元数量
HIDDEN1 = 32
HIDDEN2 = 16
OUT = 10

def run_mnist():
    batch_size = 10000
    h1, h2, c = HIDDEN1, HIDDEN2, OUT
    mlp = MNIST_MLP()
    mlp.build_model(batch_size=batch_size, hidden1=h1, hidden2=h2, out_classes=c)
    model_path = 'weight.npy'
    test_data = '../../mnist_data/t10k-images-idx3-ubyte'
    test_label = '../../mnist_data/t10k-labels-idx1-ubyte'
    mlp.load_data(test_data, test_label)
    mlp.load_model(model_path)

    for i in range(10):
        mlp.evaluate()

if __name__ == '__main__':
    run_mnist()
```

在上一节的 CPU 实验中，我们设置的默认 batch size 为 100，为了使读者对 DLP 的计算能力有更直观的比较和认识，本实验中将 batch size 改为 10000，即一次传入 10000 张图片，比较 DLP 和 CPU 计算的时间。因为 DLP 平台上 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。

2.2.5.6 实验运行

根据第2.2.5.1节 ~ 第2.2.5.5节的描述补全 mnist_mlp_demo.py 代码，并通过 Python 运行.py 代码。具体可以参考以下步骤。

1. **环境申请**

按照附录B说明申请实验环境并登录云平台，本实验的代码存放在云平台/opt/code_chap_2_3/code_ch目录下。

```
# 登录云平台
ssh root@xxx.xxx.xxx.xxx -p xxxxx
# 进入 code_chap_2_3_student 目录
cd /opt/code_chap_2_3/code_chap_2_3_student
# 初始化环境
source env.sh

```

2. 代码实现

补全 stu_upload 中的 layers_1.py，mnist_mlp_cpu.py，mnist_mlp_demo.py 文件。对 mnist_mlp_cpu.py 文件，将 build_mnist_mlp() 函数修改为以下内容：

```
def build_mnist_mlp(param_dir='weight.npy'):
    h1, h2, e = 32, 16, 10
    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(param_dir)
    return mlp

```

```
# 进入实验目录
cd exp_2_2_mnist_mlp_dlp
# 补全实验代码
vim stu_upload/layers_1.py
vim stu_upload/mnist_mlp_cpu.py
vim stu_upload/mnist_mlp_demo.py

```

3. 运行实验

复制实验 2.1中训练得到的参数复制到 stu_upload 目录下，并将参数文件重命名为 weight.npy。

```
# 运行完整实验
python main_exp_2_2.py

```

2.2.6 实验评估

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，进行一次前向传播的时间。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：修改网络隐藏层神经元的数量，运行实验 2.1 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：修改网络隐藏层神经元的数量，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

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?

参考文献

[1] 陈云霁, 李玲, 李威, 等. 智能计算系统[M]. 1rd. 机械工业出版社, 2020.

[2] LECUN Y, CORTES C, BURGES C J. The mnist database of handwritten digits[EB/OL]. http://yann.lecun.com/exdb/mnist/.

[3] ZHANG X, LIU S, ZHANG R, et al. Fixed-point back-propagation training[C]//2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). 2020.

[4] SIMONYAN K, ZISSERMAN A. Very deep convolutional networks for large-scale image recognition[C]// International Conference on Learning Representations (ICLR). 2015.

[5] DENG J, DONG W, SOCHER R, et al. ImageNet: A large-scale hierarchical image database[C]//Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR). 2009: 248-255.

[6] VEDALDI A, LENC K. Matconvnet – convolutional neural networks for matlab[C]//Proceeding of the ACM Int. Conf. on Multimedia. 2015.

[7] GATYS L A, ECKER A S, BETHGE M. Image style transfer using convolutional neural networks[C]// Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR). 2016: 2414-2423.

[8] KINGMA D P, BA J. Adam: A method for stochastic optimization[C]//International Conference on Learning Representations. 2015.

[9] ABADI M, AGARWAL A, BARHAM P, et al. TensorFlow: Large-scale machine learning on heterogeneous distributed systems[J]. arXiv preprint arXiv:1603.04467v2, 2016.

[10] ABADI M, BARHAM P, CHEN J, et al. Tensorflow: A system for large-scale machine learning[C/OL]// OSDI'16: Proceedings of the 12th USENIX symposium on operating systems design and implementation (OSDI). Berkeley, CA, USA: USENIX Association, 2016: 265-283. http://dl.acm.org/citation.cfm?id=3026877.3026899.

[11] VINCENT DUMOULIN F V. A guide to convolution arithmetic for deep learning[J]. arXiv preprint arXiv:1603.07285v2, 2018.

[12] scipy.io[EB/OL]. https://docs.scipy.org/doc/scipy/reference/tutorial/io.html.

[13] scipy.misc[EB/OL]. https://docs.scipy.org/doc/scipy-0.18.1/reference/misc.html.

[14] scipy.io.loadmat[EB/OL]. https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.loadmat.html#scipy.io.loadmat.

[15] JOHNSON J, ALAHI A, LI F F. Perceptual losses for real-time style transfer and super-resolution[C]// Proceedings of the European conference on Computer Vision. Springer, 2016: 694-711.

[16] IOFFE S, SZEGEDY C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[J]. 2015, 37:448-456.

[17] JOHNSON J, ALAHI A, LI F F. Perceptual losses for real-time style transfer and super-resolution：supplementary material[J/OL]. https://cs.stanford.edu/people/jcjohns/papers/fast-style/fast-style-supp.pdf.

[18] HE K, ZHANG X, REN S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2016: 770-778.

[19] Zeiler M D, Krishnan D, Taylor G W, et al. Deconvolutional networks[C]//IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR). 2010: 2528-2535.

[20] DMITRY ULYANOV V L, Andrea Vedaldi. Instance normalization:the missing ingredient for fast stylization [J]. arXiv preprint arXiv:1607.08022v3, 2017.

[21] VGG16 预训练模型[EB/OL]. http://www.vlfeat.org/matconvnet/models/beta16/imagenet-vgg-verydeep-16.mat.

[22] MAHENDRAN A, VEDALDI A. Understanding deep image representations by inverting them[J]. arXiv preprint arXiv:1412.0035v1, 2014.

[23] ENGSTROM L. Fast style transfer[EB/OL]. 2016. https://github.com/lengstrom/fast-style-transfer.

[24] REDMON J, FARHADI A. Yolov3: An incremental improvement[J]. arXiv preprint arXiv:1804.02767, 2018.

[25] ZHOU X, YAO C, WEN H, et al. East: an efficient and accurate scene text detector[C]//Proceedings of the IEEE conference on Computer Vision and Pattern Recognition. 2017: 5551-5560.

[26] DEVLIN J, CHANG M W, LEE K, et al. Bert: Pre-training of deep bidirectional transformers for language understanding[J]. arXiv preprint arXiv:1810.04805, 2018.

[27] XU B, WANG N, CHEN T, et al. Empirical evaluation of rectified activations in convolutional network[J]. arXiv preprint arXiv:1505.00853, 2015.

[28] NEUBECK A, VAN GOOL L. Efficient non-maximum suppression[C]//18th International Conference on Pattern Recognition (ICPR): volume 3. IEEE, 2006: 850-855.