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

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。

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**??**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。

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。

```python
# 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。

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。

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：

```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

```python
# 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

- 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？