

智能计算系统 实验教程

**智能计算系统课程团队
中国科学院计算技术研究所**

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 x ，维度为 m ；输出为一维向量 y ，维度为 n ；权重 W 是二维矩阵，维度为 $m \times n$ ，偏置 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。

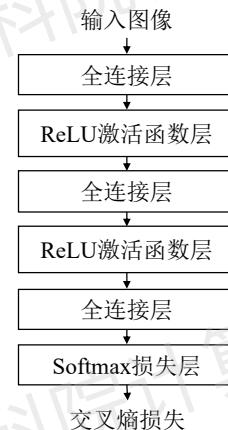


图 2.1 用于手写数字分类的三层全连接神经网络

算公式为

$$\mathbf{y} = \mathbf{W}^T \mathbf{x} + \mathbf{b} \quad (2.1)$$

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 \mathbf{y} 的偏导 $\nabla_{\mathbf{y}} L = \frac{\partial L}{\partial \mathbf{y}}$, 其维度与全连接层的输出 \mathbf{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 $\nabla_{\mathbf{W}} L = \frac{\partial L}{\partial \mathbf{W}}$ 、 $\nabla_{\mathbf{b}} L = \frac{\partial L}{\partial \mathbf{b}}$ 以及损失函数对输入的偏导 $\nabla_{\mathbf{x}} L = \frac{\partial L}{\partial \mathbf{x}}$ 计算公式分别为:

$$\begin{aligned} \nabla_{\mathbf{W}} L &= \mathbf{x} \nabla_{\mathbf{y}} L^T \\ \nabla_{\mathbf{b}} L &= \nabla_{\mathbf{y}} L \\ \nabla_{\mathbf{x}} L &= \mathbf{W}^T \nabla_{\mathbf{y}} L \end{aligned} \quad (2.2)$$

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 \mathbf{X} , 维度为 $p \times m$, 每行代表一个样本。输出也变为二维矩阵 \mathbf{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

$$\mathbf{Y} = \mathbf{X}\mathbf{W} + \mathbf{b} \quad (2.3)$$

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

$$\begin{aligned} \nabla_{\mathbf{W}} L &= \mathbf{X}^T \nabla_{\mathbf{Y}} L \\ \nabla_{\mathbf{b}} L &= \mathbf{1} \nabla_{\mathbf{Y}} L \\ \nabla_{\mathbf{x}} L &= \nabla_{\mathbf{Y}} L \mathbf{W}^T \end{aligned} \quad (2.4)$$

其中计算偏置的梯度 $\nabla_{\mathbf{b}} L$ 时, 为确保维度正确, 用 $\nabla_{\mathbf{Y}} L$ 与维度为 $1 \times p$ 的全 1 向量 $\mathbf{1}$ 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 \mathbf{y} 的维度与输入向量 \mathbf{x} 的维度相同^①。在前向传播中, 如果输入 \mathbf{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

$$y(i) = \max(0, x(i)) \quad (2.5)$$

其中 $x(i)$ 和 $y(i)$ 分别代表 x 和 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 $\nabla_y L$ 计算损失函数对输入的偏导 $\nabla_x L$ 。设 i 代表输入 x 的某个位置，则损失函数对本层的第 i 个输入的偏导 $\nabla_{x(i)} L$ 的计算公式为

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

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 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 x 计算 e 指数并进行归一化，即得到 Softmax 分类概率。假设 x 对应 i 位置的值为 $x(i)$ ， $\hat{y}(i)$ 为 i 位置的 Softmax 分类概率， $i \in [1, k]$ 且为整数，则 $\hat{y}(i)$ 的计算公式为

$$\hat{y}(i) = \frac{e^{x(i)}}{\sum_j e^{x(j)}} \quad (2.7)$$

在前向计算时，对 Softmax 分类概率 \hat{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） y 计算总的损失函数值。在分类任务中，标记 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

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

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 $\nabla_x L$ 的计算公式为

$$\nabla_x L = \frac{\partial L}{\partial x} = \hat{y} - y \quad (2.9)$$

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 X ，维度为 $p \times k$ ， 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

$$\hat{Y}(i, j) = \frac{e^{X(i, j)}}{\sum_j e^{X(i, j)}} \quad (2.10)$$

其中 $X(i, j)$ 代表 X 中对应第 i 样本 j 位置的值。当 $X(i, j)$ 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 $\hat{Y}(i, j)$ 的计算公式变为

$$\hat{Y}(i, j) = \frac{e^{X(i, j) - \max_n X(i, n)}}{\sum_j e^{X(i, j) - \max_n X(i, n)}} \quad (2.11)$$

在前向计算时, 对 Softmax 分类概率 $\hat{Y}(i, j)$ 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 \mathbf{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

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

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 $\nabla_{\mathbf{x}} L$ 的计算公式(2.9)变为:

$$\nabla_{\mathbf{x}} L = \frac{1}{p} (\hat{\mathbf{Y}} - \mathbf{Y}) \quad (2.13)$$

类似地, 损失 $\nabla_{\mathbf{x}} L$ 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 \mathbf{W} 的计算公式为

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} L \quad (2.14)$$

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

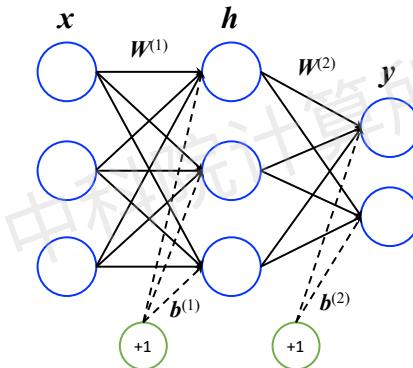


图 2.2 两层神经网络示例

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 $\mathbf{W}^{(1)}$, 偏置为 $\mathbf{b}^{(1)}$,

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

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

其中 \mathbf{h}, \mathbf{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:

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

反向传播的逐层计算公式为

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

其中 $\nabla_{\mathbf{z}} L$ 、 $\nabla_{\mathbf{h}} L$ 、 $\nabla_{\mathbf{x}} L$ 分别是损失函数对 Softmax 层、第二层、第一层的偏导， $\nabla_{\mathbf{W}^{(2)}} L$ 、 $\nabla_{\mathbf{b}^{(2)}} L$ 、 $\nabla_{\mathbf{W}^{(1)}} L$ 、 $\nabla_{\mathbf{b}^{(1)}} L$ 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

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

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 $N = 10000$), \mathbf{p}_i 为神经网络输出的第 i 张图像的推断结果, \mathbf{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

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

其中 $\mathbf{1}(\text{argmax}(\mathbf{p}_i) = y_i)$ 表示当 \mathbf{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定的，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

```
1 # file: mnist_mlp_cpu.py
2 def load_mnist(self, file_dir, is_images = 'True'):
3     bin_file = open(file_dir, 'rb')
4     bin_data = bin_file.read()
5     bin_file.close()
6     if is_images: # 读取图像数据
7         fmt_header = '>iiii'
8         magic, num_images, num_rows, num_cols = struct.unpack_from(fmt_header, bin_data,
9             0)
10    else: # 读取标记数据
11        fmt_header = '>ii'
12        magic, num_images = struct.unpack_from(fmt_header, bin_data, 0)
13        num_rows, num_cols = 1, 1
14        data_size = num_images * num_rows * num_cols
15        mat_data = struct.unpack_from('>' + str(data_size) + 'B', bin_data, struct.calcsize(
16            fmt_header))
16        mat_data = np.reshape(mat_data, [num_images, num_rows * num_cols])
17        return mat_data
```

图 2.3 MNIST 数据集文件的读取和预处理

```
1 # file: mnist_mlp_cpu.py
2 def load_data(self):
3     # 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
4     train_images = self.load_mnist(os.path.join(MNIST_DIR, TRAIN_DATA), True)
5     train_labels = _____
6     test_images = _____
7     test_labels = _____
8     self.train_data = np.append(train_images, train_labels, axis=1)
9     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 \mathbf{W} 的维度为 $m \times n$ ，偏置 \mathbf{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 $\nabla_{\mathbf{y}} L$ ，利用矩阵相乘计算权重和偏置的梯度 $\nabla_{\mathbf{W}} L$ 、 $\nabla_{\mathbf{b}} L$ 以及损失函数对本层输入的偏导 $\nabla_{\mathbf{x}} L$ 。
- 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 $\nabla_{\mathbf{W}} L$ 和偏置梯度 $\nabla_{\mathbf{b}} L$ 对本层的权重 \mathbf{W} 和偏置 \mathbf{b} 进行更新：

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} L \quad (2.20)$$

$$\mathbf{b} \leftarrow \mathbf{b} - \eta \nabla_{\mathbf{b}} L \quad (2.21)$$

- 参数加载：从该函数的输入中读取本层的权重 \mathbf{W} 和偏置 \mathbf{b} 。
- 参数保存：返回本层当前的权重 \mathbf{W} 和偏置 \mathbf{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 \mathbf{y} 中对应位置的值置为 0。

```
1 # file: layers_1.py
2 class FullyConnectedLayer(object):
3     def __init__(self, num_input, num_output): # 全连接层初始化
4         self.num_input = num_input
5         self.num_output = num_output
6     def init_param(self, std=0.01): # 参数初始化
7         self.weight = np.random.normal(loc=0.0, scale=std, size=(self.num_input, self.
8             num_output))
9         self.bias = np.zeros([1, self.num_output])
10    def forward(self, input): # 前向传播的计算
11        self.input = input
12        # TODO: 全连接层的前向传播, 计算输出结果
13        self.output = _____
14        return self.output
15    def backward(self, top_diff): # 反向传播的计算
16        # TODO: 全连接层的反向传播, 计算参数梯度和本层损失
17        self.d_weight = _____
18        self.d_bias = _____
19        bottom_diff = _____
20        return bottom_diff
21    def update_param(self, lr): # 参数更新
22        # TODO: 利用梯度对全连接层参数进行更新
23        self.weight = _____
24        self.bias = _____
25    def load_param(self, weight, bias): # 参数加载
26        self.weight = weight
27        self.bias = bias
28    def save_param(self): # 参数保存
29        return self.weight, self.bias
```

图 2.5 全连接层的实现示例

```
1 # file: layers_1.py
2 class ReLULayer(object):
3     def forward(self, input): # 前向传播的计算
4         self.input = input
5         # TODO: ReLU 层的前向传播, 计算输出结果
6         output = _____
7         return output
8     def backward(self, top_diff): # 反向传播的计算
9         # TODO: ReLU 层的反向传播, 计算本层损失
10        bottom_diff = _____
11        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。

```

1 # file: layers_1.py
2 class SoftmaxLossLayer(object):
3     def forward(self, input): # 前向传播的计算
4         # TODO: Softmax 损失层的前向传播, 计算输出结果
5         input_max = np.max(input, axis=1, keepdims=True)
6         input_exp = np.exp(input - input_max)
7         self.prob = _____
8         return self.prob
9     def get_loss(self, label): # 计算损失
10        self.batch_size = self.prob.shape[0]
11        self.label_onehot = np.zeros_like(self.prob)
12        self.label_onehot[np.arange(self.batch_size), label] = 1.0
13        loss = -np.sum(np.log(self.prob) * self.label_onehot) / self.batch_size
14        return loss
15    def backward(self): # 反向传播的计算
16        # TODO: Softmax 损失层的反向传播, 计算本层损失
17        bottom_diff = _____
18        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。

```
1 # file: mnist_mlp_cpu.py
2 class MNIST_MLP(object):
3     def __init__(self, batch_size=100, input_size=784, hidden1=32, hidden2=16,
4                  out_classes=10, lr=0.01, max_epoch=2, print_iter=100):
5         # 神经网络初始化
6         self.batch_size = batch_size
7         self.input_size = input_size
8         self.hidden1 = hidden1
9         self.hidden2 = hidden2
10        self.out_classes = out_classes
11        self.lr = lr
12        self.max_epoch = max_epoch
13        self.print_iter = print_iter
14    def build_model(self): # 建立网络结构
15        # TODO: 建立三层神经网络结构
16        self.fc1 = FullyConnectedLayer(self.input_size, self.hidden1)
17        self.relu1 = ReLULayer()
18
19        self.fc3 = FullyConnectedLayer(self.hidden2, self.out_classes)
20        self.softmax = SoftmaxLossLayer()
21        self.update_layer_list = [self.fc1, self.fc2, self.fc3]
22    def init_model(self): # 神经网络参数初始化
23        for layer in self.update_layer_list:
24            layer.init_param()
```

图 2.8 三层神经网络的网络结构模块实现示例

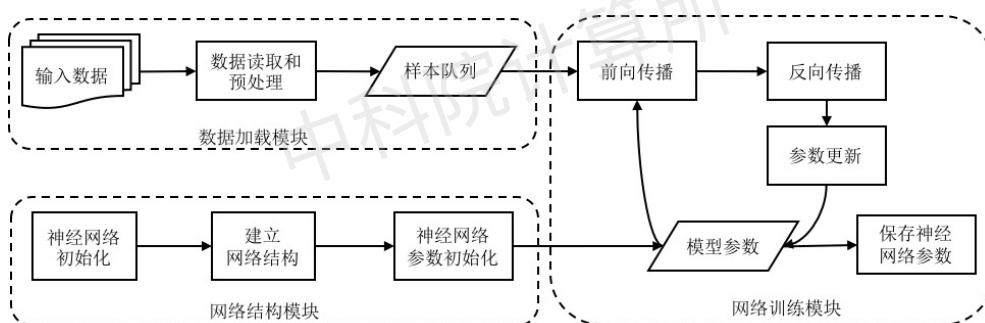


图 2.9 神经网络训练流程

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

```

1 # file: mnist_mlp_cpu.py
2 def forward(self, input): # 神经网络的前向传播
3     # TODO: 神经网络的前向传播
4     h1 = self.fc1.forward(input)
5     h1 = self.relu1.forward(h1)
6
7     prob = self.softmax.forward(h3)
8     return prob
9
10 def backward(self): # 神经网络的反向传播
11     # TODO: 神经网络的反向传播
12     dloss = self.softmax.backward()
13
14     dh1 = self.relu1.backward(dh2)
15     dh1 = self.fc1.backward(dh1)
16
17 def update(self, lr): # 神经网络参数更新
18     for layer in self.update_layer_list:
19         layer.update_param(lr)
20
21 def save_model(self, param_dir): # 保存神经网络参数
22     params = {}
23     params['w1'] = self.fc1.save_param()
24     params['b2'] = self.fc2.save_param()
25     params['w3'] = self.fc3.save_param()
26     np.save(param_dir, params)
27
28 def train(self): # 训练函数主体
29     max_batch = self.train_data.shape[0] / self.batch_size
30     for idx_epoch in range(self.max_epoch):
31         mlp.shuffle_data()
32         for idx_batch in range(max_batch):
33             batch_images = self.train_data[idx_batch * self.batch_size:(idx_batch + 1) * self.
34             batch_size, :-1]
35             batch_labels = self.train_data[idx_batch * self.batch_size:(idx_batch + 1) * self.
36             batch_size, -1]
37             prob = self.forward(batch_images)
38             loss = self.softmax.get_loss(batch_labels)
39             self.backward()
40             self.update(self.lr)
41             if idx_batch % self.print_iter == 0:
42                 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

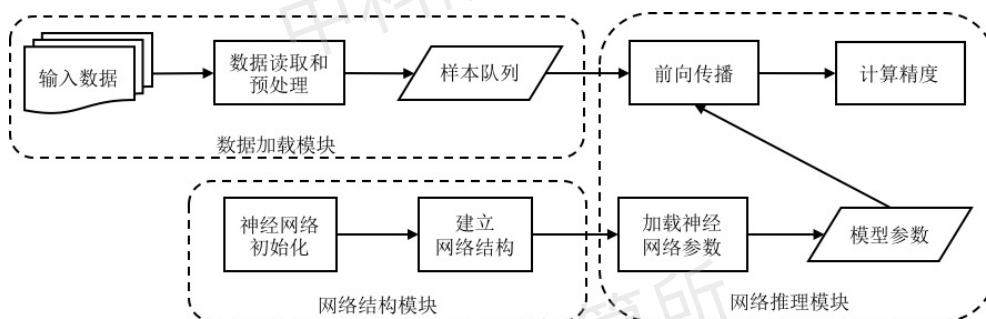


图 2.11 神经网络推断流程

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。

```

1 # file: mnist_mlp_cpu.py
2 def load_model(self, param_dir): # 加载神经网络参数
3     params = np.load(param_dir).item()
4     self.fc1.load_param(params['w1'], params['b1'])
5     self.fc2.load_param(params['w2'], params['b2'])
6     self.fc3.load_param(params['w3'], params['b3'])
7
8 def evaluate(self): # 推断函数主体
9     pred_results = np.zeros([self.test_data.shape[0]])
10    for idx in range(self.test_data.shape[0]/self.batch_size):
11        batch_images = self.test_data[idx*self.batch_size:(idx+1)*self.batch_size, :-1]
12        prob = self.forward(batch_images)
13        pred_labels = np.argmax(prob, axis=1)
14        pred_results[idx*self.batch_size:(idx+1)*self.batch_size] = pred_labels
15    accuracy = np.mean(pred_results == self.test_data[:, -1])
16    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。

```

1 # file: mnist_mlp_cpu.py
2 def build_mnist_mlp(param_dir='weight.npy'):
3     h1, h2, e = 32, 16, 10
4     mlp = MNIST_MLP(hidden1=h1, hidden2=h2, max_epoch=e)
5     mlp.load_data()
6     mlp.build_model()
7     mlp.init_model()
8     mlp.train()
9     mlp.save_model('mlp-%d-%d-%depoch.npy' % (h1, h2, e))
10    # mlp.load_model('mlp-%d-%d-%depoch.npy' % (h1, h2, e))
11    return mlp
12
13 if __name__ == '__main__':
14     mlp = build_mnist_mlp()
15     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_chap_2_3目录下。

```
1 # 登录云平台
2 ssh root@xxx.xxx.xxx.xxx -pxxxxx
3 # 进入 code_chap_2_3_student 目录
4 cd /opt/code_chap_2_3/code_chap_2_3_student
5 # 初始化环境
6 source env.sh
7
```

2. 代码实现

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

```
1 # 进入实验目录
2 cd exp_2_1_mnist_mlp
3 # 补全 layers_1.py, mnist_mlp_cpu.py
4 vim stu_upload/layers_1.py
5 vim stu_upload/mnist_mlp_cpu.py
6
```

3. 运行实验

```
1 # 运行完整实验
2 python main_exp_2_1.py
3
```

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

$$q_x = \text{round}\left(\frac{r_x}{2^{\textit{position}}}\right)$$

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
<i>position</i>	小数点的位置
<i>scale</i>	缩放系数
<i>offset</i>	偏移量
<i>round</i>	四舍五入取整
<i>n</i>	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

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

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

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：

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

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

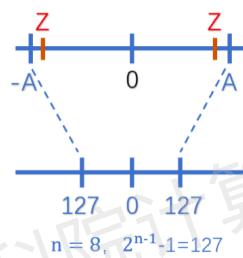


图 2.14 int8 量化

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 pycnml.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。pycnml.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。

```

1 # 实例化 CnmlNet
2 net = pycnml.CnmlNet()
3 # 设定网络输入维度
4 net.setInputShape(1, 3, 224, 224)
5 # conv1_1
6 # 创建卷积和全连接层时需要输入量化参数
7 net.createConvLayer('conv1_1', 64, 3, 1, 1, 1, input_quant_params[0])
8 # relu1_1
9 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: pycnml.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: pycnml.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: pycnml.CnmlNet().createMlpLayer(input_shape, out_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: pycnml.CnmlNet().createReLuLayer(input_shape)	创建 ReLu 激活函数层	input_shape (list) : 输入数据的形状
createSoftmaxLayer: pycnml.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: pycnml.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: pycnml.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: pycnml.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: pycnml.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。

```

1 # file: mnist_mlp_demo.py
2 def load_data(self, data_path, label_path):
3     # 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
4     test_images = _____
5     test_labels = _____
6     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：

```

1 # file: mnist_mlp_demo.py
2 class MNIST_MLP(object):
3     def __init__(self):
4         # 初始化网络，创建pycnml.CnmlNet() 实例
5         self.net = pycnml.CnmlNet()
6         self.input_quant_params = [] #输入数据的量化参数
7         self.filter_quant_params = [] #模型参数的量化参数
8
9     def build_model(self, batch_size=100, input_size=784,
10                 hidden1=32, hidden2=16, out_classes=10,
11                 quant_param_path='./data/mnist_mlp_data/mnist_mlp_quant_param.npz'):
12         # 使用 pycnml 的接口建立三层神经网络结构
13         self.batch_size = batch_size
14         self.out_classes = out_classes
15
16         # 读取量化参数
17         params = np.load(quant_param_path)
18         input_params = params['input']
19         filter_params = params['filter']
20         for i in range(0, len(input_params), 2):
21             self.input_quant_params.append(pycnml.QuantParam(int(input_params[i]), float(
22             input_params[i+1])))
23             for i in range(0, len(filter_params), 2):
24                 self.filter_quant_params.append(pycnml.QuantParam(int(filter_params[i]), float(
25                 filter_params[i+1])))
26
27         # 创建神经网络的层
28         self.net.setInputShape(batch_size, input_size, 1, 1)
29         # TODO: 使用 pycnml 搭建三层神经网络结构
30         # fc1
31         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：

```

1 # file: mnist_mlp_demo.py
2 def load_model(self, param_dir):
3     # TODO: 分别为三层全连接层加载参数
4     params = np.load(param_dir).item()
5     weigh1 = np.transpose(params['w1'], [1, 0]).flatten().astype(np.float)
6     bias1 = params['b1'].flatten().astype(np.float)
7     self.net.loadParams(0, weigh1, bias1, self.filter_quant_params[0])
8     weigh2 = np.transpose(params['w2'], [1, 0]).flatten().astype(np.float)
9     bias2 = params['b2'].flatten().astype(np.float)
10    -----
11    weigh3 = np.transpose(params['w3'], [1, 0]).flatten().astype(np.float)
12    bias3 = params['b3'].flatten().astype(np.float)
13    -----
14
15 def forward(self): # 前向传播
16     return self.net.forward()
17
18 def evaluate(self):
19     pred_results = np.zeros([self.test_data.shape[0]])
20     # 读取一定批量的测试数据进行前向传播
21     for idx in range(self.test_data.shape[0]/self.batch_size):
22         batch_images = self.test_data[idx*self.batch_size:(idx+1)*self.batch_size, :-1]
23         data = batch_images.flatten().tolist()
24         # 加载输入数据
25         self.net.setInputData(data)
26         # 打印推理的时间
27         start = time.time()
28         self.forward()
29         end = time.time()
30         print('inference time: %f' % (end - start))
31         prob = self.net.getOutputData()
32         prob = np.array(prob).reshape((self.batch_size, self.out_classes))
33         pred_labels = np.argmax(prob, axis=1)
34         pred_results[idx*self.batch_size:(idx+1)*self.batch_size] = pred_labels
35     accuracy = np.mean(pred_results == self.test_data[:, -1])
36     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，pycmml 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 pycmml 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：

```
1 # file: mnist_mlp_demo.py
2 # 神经元数量
3 HIDDEN1 = 32
4 HIDDEN2 = 16
5 OUT = 10
6
7 def run_mnist():
8     batch_size = 10000
9     h1, h2, c = HIDDEN1, HIDDEN2, OUT
10    mlp = MNIST_MLP()
11    mlp.build_model(batch_size=batch_size, hidden1=h1, hidden2=h2, out_classes=c)
12    model_path = 'weight.npy'
13    test_data = '../.../mnist_data/t10k-images-idx3-ubyte'
14    test_label = '../.../mnist_data/t10k-labels-idx1-ubyte'
15    mlp.load_data(test_data, test_label)
16    mlp.load_model(model_path)
17
18    for i in range(10):
19        mlp.evaluate()
20
21 if __name__ == '__main__':
22     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_chap_2_3目录下。

```

1 # 登录云平台
2 ssh root@xxx.xxx.xxx.xxx -pxxxxx
3 # 进入 code_chap_2_3_student 目录
4 cd /opt/code_chap_2_3/code_chap_2_3_student
5 # 初始化环境
6 source env.sh
7

```

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:

```

1 def build_mnist_mlp(param_dir='weight.npy'):
2     h1, h2, e = 32, 16, 10
3     mlp = MNIST_MLP(hidden1=h1, hidden2=h2, max_epoch=e)
4     mlp.load_data()
5     mlp.build_model()
6     mlp.init_model()
7     # mlp.train()
8     # mlp.save_model('mlp-%d-%d-%depoch.npy' % (h1, h2, e))
9     mlp.load_model(param_dir)
10    return mlp
11

```

```

1 # 进入实验目录
2 cd exp_2_2_mnist_mlp_dlp
3 # 补全实验代码
4 vim stu_upload/layers_1.py
5 vim stu_upload/mnist_mlp_cpu.py
6 vim stu_upload/mnist_mlp_demo.py
7

```

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。

```

1 # 运行完整实验
2 python main_exp_2_2.py
3

```

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？

第3章 深度学习应用实验

《智能计算系统》课程教材中采用图像风格迁移作为驱动范例，将深度学习算法、编程框架、深度学习处理器硬件架构贯穿起来。图像风格迁移是深度学习的一个典型应用，有非实时和实时两种实现方式，其中非实时风格迁移算法使用 VGG19 作为核心网络结构。本章首先介绍如何用 Python 语言实现基于 VGG19 的图像分类，然后介绍如何在 DLP 上实现图像分类，最后介绍如何用 Python 语言实现非实时风格迁移算法。本章的三个实验，均是基于第2章的实验框架扩展而来，同时会复用第2章实验中的全连接层、ReLU 激活函数层等基本单元，在此基础上加入新的基本单元如卷积层、最大池化层。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）中深度学习部分的知识点。

3.1 基于 VGG19 实现图像分类

3.1.1 实验目的

掌握卷积神经网络的设计原理，掌握卷积神经网络的使用方法，能够使用 Python 语言实现 VGG19^[4] 网络模型对给定的输入图像进行分类。具体包括：

- 1) 加深对深度卷积神经网络中卷积层、最大池化层等基本单元的理解。
- 2) 利用 Python 语言实现 VGG19 的前向传播计算，加深对 VGG19 网络结构的理解，为后续风格迁移中使用 VGG19 网络进行特征提取奠定基础。
- 3) 在第2.1节实验的基础上将三层神经网络扩展为 VGG19 网络，加深对神经网络工程实现中基本模块演变的理解，为后续建立更复杂的综合实验（如风格迁移）奠定基础。

实验工作量：约 30 行代码，约需 3 个小时。

3.1.2 背景介绍

3.1.2.1 卷积神经网络中的基本单元

常见的卷积神经网络结构如图3.1所示。卷积层后面会使用 ReLU 等激活函数，N 个卷积层后通常会使用一个最大池化层（也有使用平均池化的）；卷积和池化组合出现 M 次之后，提取出来的卷积特征会经过 K 个全连接层映射到若干个输出特征上，最后再经过一个全连接层或 Softmax 层来决定最终的输出。在第2.1节实验中，已经介绍了全连接层、ReLU 激活函数、Softmax 层，本节介绍本实验中新增的基本单元：卷积层和最大池化层。更多关于卷积层和最大池化层的介绍详见《智能计算系统》课程教材第 3.1 节。

卷积层

与全连接层类似，卷积层中的参数包括权重（即卷积核参数）和偏置。VGG19 中使用的都是多输入输出特征图的卷积运算。假设输入特征图 X 的维度为 $N \times C_{in} \times H_{in} \times W_{in}$ ，其中 N 是输入的样本个数（在本实验中 $N = 1$ ）， C_{in} 是输入的通道数， H_{in} 和 W_{in} 是输入特征图的高和宽。卷积核张量 W 用四维矩阵表示，维度为 $C_{in} \times K \times K \times C_{out}$ ，其中 $K \times K$ 为

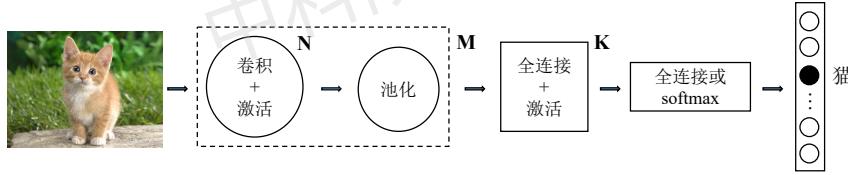


图 3.1 常见卷积神经网络结构

卷积核的高度 \times 宽度， C_{out} 为输出特征图的通道数。卷积层的偏置 \mathbf{b} 用一维向量表示，维度为 C_{out} 。同时定义输入特征图的边界扩充大小 p 、卷积步长 s 。输出特征图 \mathbf{Y} 由输入 \mathbf{X} 与卷积核 \mathbf{W} 内积并加偏置 \mathbf{b} 计算得到， \mathbf{Y} 的维度为 $N \times C_{out} \times H_{out} \times W_{out}$ ，其中 H_{out} 和 W_{out} 是输出特征图的高和宽。

前向传播计算时，为了保证卷积之后的有效输出尺寸与输入尺寸一致，首先对卷积层的输入 \mathbf{X} 做边界扩充 (padding)，即在输入特征图的上下以及左右边界分别增加 p 行以及 p 列的 0。维度为 $N \times C_{in} \times H_{in} \times W_{in}$ 的输入特征图，经过大小为 p 的边界扩充，得到扩充后的特征图 \mathbf{X}_{pad}

$$\mathbf{X}_{pad}(n, c_{in}, h, w) = \begin{cases} \mathbf{X}(n, c_{in}, h - p, w - p) & p \leq h < p + H_{in}, p \leq w < p + W_{in} \\ 0 & \text{其他} \end{cases} \quad (3.1)$$

其中 $n \in [1, N]$ 、 $c_{in} \in [1, C_{in}]$ 、 $h \in [1, H_{in}]$ 、 $w \in [1, W_{in}]$ 分别表示输入特征图的样本号、通道号、行号、列号，均为整数。 \mathbf{X}_{pad} 的维度为 $N \times C_{in} \times H_{pad} \times W_{pad}$ ，其中高度 H_{pad} 和宽度 W_{pad} 分别为

$$H_{pad} = H_{in} + 2p \quad W_{pad} = W_{in} + 2p \quad (3.2)$$

然后，用边界扩充后的特征图与卷积核做矩阵内积并与偏置相加得到输出特征图 \mathbf{Y}

$$\mathbf{Y}(n, c_{out}, h, w) = \sum_{c_{in}, k_h, k_w} \mathbf{W}(c_{in}, k_h, k_w, c_{out}) \mathbf{X}_{pad}(n, c_{in}, hs + k_h, ws + k_w) + \mathbf{b}(c_{out}) \quad (3.3)$$

其中 $n \in [1, N]$ 、 $c_{out} \in [1, C_{out}]$ 、 $h \in [1, H_{out}]$ 、 $w \in [1, W_{out}]$ 分别表示输出特征图的样本号、通道号、行号、列号； $k_h \in [1, K]$ 、 $k_w \in [1, K]$ 表示卷积核的行号和列号； $c_{in} \in [1, C_{in}]$ 表示输入特征图的通道号。这些符号的值均为整数。输出特征图 \mathbf{Y} 的高度和宽度分别是

$$\begin{aligned} H_{out} &= \left\lfloor \frac{H_{pad} - K}{s} + 1 \right\rfloor = \left\lfloor \frac{H_{in} + 2p - K}{s} + 1 \right\rfloor \\ W_{out} &= \left\lfloor \frac{W_{pad} - K}{s} + 1 \right\rfloor = \left\lfloor \frac{W_{in} + 2p - K}{s} + 1 \right\rfloor \end{aligned} \quad (3.4)$$

反向传播计算时，假设损失函数为 L ，给定损失函数对本层输出的偏导 $\nabla_{\mathbf{Y}} L$ ，其维度与卷积层的输出特征图相同，均为 $N \times C_{out} \times H_{out} \times W_{out}$ 。根据链式法则，可以计算权重和

偏置的梯度 $\nabla_{\mathbf{W}} L$ 、 $\nabla_{\mathbf{b}} L$ 以及损失函数对边界扩充后的输入的偏导 $\nabla_{\mathbf{X}_{pad}} L$ ，计算公式为

$$\begin{aligned}\nabla_{\mathbf{W}(c_{in}, k_h, k_w, c_{out})} L &= \sum_{n, h, w} \nabla_{\mathbf{Y}(n, c_{out}, h, w)} L \ X_{pad}(n, c_{in}, hs + k_h, ws + k_w) \\ \nabla_{\mathbf{b}(j)} L &= \sum_{n, h, w} \nabla_{\mathbf{Y}(n, c_{out}, h, w)} L \\ \nabla_{\mathbf{X}_{pad}(n, c_{in}, hs + k_h, ws + k_w)} L &= \sum_j \nabla_{\mathbf{Y}(n, c_{out}, h, w)} L \ \mathbf{W}(c_{in}, k_h, k_w, c_{out})\end{aligned}\quad (3.5)$$

之后剪裁掉 $\nabla_{\mathbf{X}_{pad}} L$ 中扩充的边界，得到本层的 $\nabla_{\mathbf{X}} L$ ，计算公式为

$$\nabla_{\mathbf{X}} L(n, c_{in}, h, w) = \nabla_{\mathbf{X}_{pad}} L(n, c_{in}, h + p, w + p) \quad (3.6)$$

其中 $n \in [1, N]$, $c_{in} \in [1, C_{in}]$, $h \in [1, H_{in}]$, $w \in [1, W_{in}]$ 。

最大池化层

假设最大池化层的输入特征图 \mathbf{X} 的维度为 $N \times C \times H_{in} \times W_{in}$ ，其中 N 是输入的样本个数（在本实验中 $N = 1$ ）， C 是输入的通道数， H_{in} 和 W_{in} 是输入特征图的高和宽。池化窗口的高和宽均为 K ，池化步长为 s ，输出特征图 \mathbf{Y} 的维度为 $N \times C \times H_{out} \times W_{out}$ ，其中 H_{out} 和 W_{out} 是输出特征图的高和宽。

前向传播计算时，输出特征图 \mathbf{Y} 中某一位置的值是输入特征图 \mathbf{X} 的对应池化窗口内的最大值，计算公式为

$$\mathbf{Y}(n, c, h, w) = \max_{k_h, k_w} X(n, c, hs + k_h, ws + k_w) \quad (3.7)$$

其中 $n \in [1, N]$ 、 $c \in [1, C]$ 、 $h \in [1, H_{out}]$ 、 $w \in [1, W_{out}]$ 分别表示输出特征图的样本号、通道号、行号、列号， $k_h \in [1, K]$ 、 $k_w \in [1, K]$ 表示池化窗口内的坐标位置，均为整数。

反向传播计算过程可以根据前向传播公式(3.7)推导获得。给定损失函数对本层输出的偏导 $\nabla_{\mathbf{Y}} L$ ，其维度与最大池化层的输出特征图相同，均为 $N \times C \times H_{out} \times W_{out}$ 。由于最大池化层在前向传播后仅保留池化窗口内的最大值，因此在反向传播时，仅将后一层损失中对该池化窗口的值传递给池化窗口内最大值所在位置，其他位置值置为 0。在反向传播时需先计算最大值所在位置 p ，计算公式为：

$$p(n, c, h, w) = \underset{k_h, k_w}{F} (X(n, c, hs + k_h, ws + k_w)) \quad (3.8)$$

其中 F 代表取最大值所在位置的函数，返回最大值位于池化窗口中的坐标向量 $p(n, c, h, w) = [q(0), q(1)]$ ，其中 $q(0)$ 对应 h 方向的坐标， $q(1)$ 对应 w 方向的坐标。 $n \in [1, N]$, $c \in [1, C]$, $h \in [1, H_{out}]$, $w \in [1, W_{out}]$, $k_h \in [1, K]$, $k_w \in [1, K]$ 均为输入输出特征图和池化窗口上的位置坐标。利用最大值所在位置 $[q(0), q(1)]$ 可得最大池化层的损失 $\nabla_{\mathbf{X}} L$ ，计算公式为：

$$\nabla_{\mathbf{X}(n, c, hs + q(0), ws + q(1))} L = \nabla_{\mathbf{Y}} L(n, c, h, w) \quad (3.9)$$

3.1.2.2 VGG19 网络的基本结构

VGG19^[4] 是经典的深度卷积神经网络结构，包含 5 个阶段共 16 个卷积层和 3 个全连接层，如表3.1所示。前 2 个阶段各有 2 个卷积层，后 3 个阶段各有 4 个卷积层。每个卷积

层均使用 3×3 大小的卷积核，边界扩充大小为 1，步长为 1，即保持输入输出特征图的高和宽不变。每个阶段的卷积层的通道数在不断变化。在每个阶段的第一个卷积层，输入通道数为上一个卷积层的输出通道数（第一个阶段的输入通道数为原始图像通道数）。5 个阶段的卷积层输出通道数分别为 64、128、256、512、512。每个阶段除第一个卷积层外，其他卷积层均保持输入和输出通道数相同。每个卷积层后面都跟随有 ReLU 层作为激活函数，每个阶段最后都跟随有一个最大池化层，将特征图的高和宽缩小为原来的 $1/2$ 。3 个全连接层中前 2 个全连接层后面也跟随有 ReLU 层。值得注意的是，第五阶段输出的特征图会进行变形，将四维特征图变形为二维矩阵作为全连接层的输入。网络最后是 Softmax 层计算分类概率。VGG19 的超参数配置详见表 3.1，注意表中省略了卷积层和全连接层后的 ReLU 层。更多关于 VGG19 网络基本结构的介绍详见《智能计算系统》课程教材第 3.2.2 节。

表 3.1 VGG19 网络基本结构

名字	类型	卷积核/池化核	步长	边界扩充	输入通道数	输出通道数	输出特征图高和宽
conv1_1	卷积层	3	1	1	3	64	224
conv1_2	卷积层	3	1	1	64	64	224
pool1	最大池化层	2	2	-	64	64	112
conv2_1	卷积层	3	1	1	64	128	112
conv2_2	卷积层	3	1	1	128	128	112
pool2	最大池化层	2	2	-	128	128	56
conv3_1	卷积层	3	1	1	128	256	56
conv3_2	卷积层	3	1	1	256	256	56
conv3_3	卷积层	3	1	1	256	256	56
conv3_4	卷积层	3	1	1	256	256	56
pool3	最大池化层	2	2	-	256	256	28
conv4_1	卷积层	3	1	1	256	512	28
conv4_2	卷积层	3	1	1	512	512	28
conv4_3	卷积层	3	1	1	512	512	28
conv4_4	卷积层	3	1	1	512	512	28
pool4	最大池化层	2	2	-	512	512	14
conv5_1	卷积层	3	1	1	512	512	14
conv5_2	卷积层	3	1	1	512	512	14
conv5_3	卷积层	3	1	1	512	512	14
conv5_4	卷积层	3	1	1	512	512	14
pool5	最大池化层	2	2	-	512	512	7
fc6	全连接层	-	-	-	25088	4096	-
fc7	全连接层	-	-	-	4096	4096	-
fc8	全连接层	-	-	-	4096	1000	-
Softmax	损失层	-	-	-	-	-	-

3.1.3 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：官方训练 VGG19 使用的数据集为 ImageNet^[5]。该数据集包括约 128 万训练图像和 5 万张测试图像，共有 1000 个不同的类别。本实验使用了官方训练好的模型参数，并不需要直接使用 ImageNet 数据集进行 VGG19 模型的训练。

3.1.4 实验内容

本实验利用 VGG19 网络进行图像分类。首先建立 VGG19 的网络结构，然后利用 VGG19 的官方模型参数对给定图像进行分类。VGG19 网络的模型参数是在 ImageNet^[5] 数据集上训练获得，其输出结果对应 ImageNet 数据集中的 1000 个类别概率。

在工程实现中，依然按照第2.1节实验的模块划分方法，每个模块的具体实现基于第2.1节实验进行改进。由于本实验只涉及 VGG19 网络的推断过程，因此本实验仅包括数据加载模块、基本单元模块、网络结构模块、网络推断模块，不包括网络训练模块。

3.1.5 实验步骤

3.1.5.1 数据加载模块

数据加载模块实现数据读取和预处理，程序示例如图3.2所示。本实验采用 ImageNet 图像数据集，该数据集以.jpg 或.png 压缩文件格式存放每张 RGB 图像，且不同图像的尺寸可能不同。为了统一神经网络输入的大小，读入图像数据后，首先需要将图像缩放到 224×224 大小，并存储在矩阵中。其次，需要对输入图像做标准化，将输入值范围从 [0,255] 标准化为均值为 0 的区间，从而提高神经网络的训练速度和稳定性。具体做法是图像的每个像素值减去 ImageNet 数据集的像素均值，该图像均值在加载 VGG19 模型参数的同时读入。本实验中使用 VGG19 模型中自带的图像均值进行输入图像标准化，是为了确保与官方使用 VGG19 网络时的预处理方式保持一致。最后，将标准化后的图像矩阵转换为神经网络输入的统一维度，即 $N \times C \times H \times W$ ，其中 N 是输入的样本数（由于图像是逐张读入的，因此 $N = 1$ ）， C 是输入的通道数（本实验输入图像是 RGB 彩色图像，因此 $C = 3$ ）， H 和 W 分别表示输入的高和宽（缩放后的图像的高和宽均为 224）。

```

1 # file: vgg_cpu.py
2 def load_image(self, image_dir):
3     print('Loading and preprocessing image from ' + image_dir)
4     self.input_image = scipy.misc.imread(image_dir)
5     self.input_image = scipy.misc.imresize(self.input_image,[224,224,3])
6     self.input_image = np.array(self.input_image).astype(np.float32)
7     self.input_image -= self.image_mean
8     self.input_image = np.reshape(self.input_image, [1]+list(self.input_image.shape))
9     # input dim [N, channel, height, width]
10    self.input_image = np.transpose(self.input_image, [0, 3, 1, 2])

```

图 3.2 VGG19 的数据加载模块实现示例

3.1.5.2 基本单元模块

本实验仅实现 VGG19 的推断过程，因此不需要实现反向传播计算和参数的更新，仅需实现层的初始化、参数初始化、前向传播计算、参数加载等基本操作。在 VGG19 网络中，

包含卷积层、ReLU 层、最大池化层、全连接层和 Softmax 层。其中全连接层、ReLU 层和 Softmax 层在第2.1节实验中已经实现，可以直接使用，本节重点介绍卷积层和池化层实现。此外还需实现一个 flatten（扁平化）层，用在 VGG19 中第一个全连接层之前，用于将最大池化层（pool5）输出的四维特征图矩阵变形为二维矩阵作为全连接层的输入。最大池化层和 flatten 层中没有参数，不包含参数初始化和参数加载操作。

卷积层

程序示例如图3.3所示，定义了以下成员函数：

- 层的初始化：需要定义卷积的超参数，包括卷积核的高（或宽） K 、输入特征图的通道数 C_{in} 、输出特征图通道数 C_{out} 、特征图边界扩充大小 p 、卷积步长 s 等。
- 参数初始化：卷积层的参数包括权重（卷积核）和偏置。与全连接层类似，通常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：根据公式(3.1)和(3.3)可进行卷积层的前向传播计算。首先利用公式(3.1)对输入特征图进行边界扩充。之后利用公式(3.3)将卷积核与边界扩充后的特征图计算矩阵内积并与偏置相加获得当前位置的输出特征图结果，将卷积核进行滑动获得整个输出特征图的结果。在工程实现中，最简单直接的实现方式是利用四重循环计算输出特征图所有位置的值。由于 VGG19 网络中的所有卷积层都是 3×3 卷积核，即 $K = 3$ ，边界扩充大小 $p = 1$ ，步长 $s = 1$ ，因此 VGG19 网络中的所有卷积层输出特征图的高和宽与输入特征图保持相同。
- 参数加载：从该函数的输入中读取本层的权重 W 和偏置 b 。

最大池化层

程序示例如图3.4所示，定义了以下成员函数：

- 层的初始化：需要定义最大池化的超参数，包括池化窗口的高（或宽） K 和池化步长 s 。
- 前向传播计算：根据公式(3.7)可计算最大池化层的前向传播结果。输出特征图的某一位置的值为输入特征图的对应池化窗口中的最大值。由于输出特征图的每个位置的值都是输入特征图的对应池化窗口的最大值，因此最简单直接的实现方式是用四重循环来计算输出特征图所有位置的值。

Flatten 层

程序示例如图3.5所示，定义了以下成员函数：

- 层的初始化：flatten 层用于改变特征图的维度，将输入特征图中每个样本的特征平铺成一个向量。初始化 flatten 层时需要定义输入特征图和输出特征图的维度。
- 前向传播计算：假设输入特征图 X 的维度为 $N \times C \times H \times W$ ，其中 N 是输入的样本个数（在本实验中 $N = 1$ ）， C 是输入的通道数， H 和 W 是输入特征图的高和宽。将输入特征图中每个样本的特征平铺成一个向量后，输出特征图的维度变为 $N \times (CHW)$ 。注意 VGG19 官方模型所使用的深度学习平台 MatConvNet^[6] 的特征图存储方式与本实验中不同。MatConvNet 中特征图维度为 $N \times H \times W \times C$ ，而本实验中特征图 X 的维度为 $N \times C \times H \times W$ 。因此为避免使用官方模型计算出现错误，flatten 层在改变输入特征图的维度前，需要将输入特征图进行维度交换，保持与 MatConvNet 的特征图存储方式一致。

```

1 # file: layer_2.py
2 class ConvolutionalLayer(object):
3     def __init__(self, kernel_size, channel_in, channel_out, padding, stride):
4         # 卷积层的初始化
5         self.kernel_size = kernel_size
6         self.channel_in = channel_in
7         self.channel_out = channel_out
8         self.padding = padding
9         self.stride = stride
10    def init_param(self, std=0.01): # 参数初始化
11        self.weight = np.random.normal(loc=0.0, scale=std, size=(self.channel_in, self.
12            kernel_size, self.kernel_size, self.channel_out))
13        self.bias = np.zeros([self.channel_out])
14    def forward(self, input): # 前向传播的计算
15        self.input = input # [N, C, H, W]
16        height = self.input.shape[2] + self.padding * 2
17        width = self.input.shape[3] + self.padding * 2
18        self.input_pad = np.zeros([self.input.shape[0], self.input.shape[1], height,
19            width])
20        self.input_pad[:, :, self.padding:self.padding+self.input.shape[2], self.padding
21            :self.padding+self.input.shape[3]] = self.input
22        height_out = (height - self.kernel_size) / self.stride + 1
23        width_out = (width - self.kernel_size) / self.stride + 1
24        self.output = np.zeros([self.input.shape[0], self.channel_out, height_out,
25            width_out])
26        for idxn in range(self.input.shape[0]):
27            for idxc in range(self.channel_out):
28                for idxh in range(height_out):
29                    for idxw in range(width_out):
30                        # TODO: 计算卷积层的前向传播, 即特征图与卷积核的内积再加偏置
31                        self.output[idxn, idxc, idxh, idxw] = _____
32        return self.output
33    def load_param(self, weight, bias): # 参数加载
34        self.weight = weight
35        self.bias = bias

```

图 3.3 卷积层的实现示例

```

1 # file: layer_2.py
2 class MaxPoolingLayer(object):
3     def __init__(self, kernel_size, stride): # 最大池化层的初始化
4         self.kernel_size = kernel_size
5         self.stride = stride
6     def forward(self, input): # 前向传播的计算
7         start_time = time.time()
8         self.input = input # [N, C, H, W]
9         self.max_index = np.zeros(self.input.shape)
10        height_out = (self.input.shape[2] - self.kernel_size) / self.stride + 1
11        width_out = (self.input.shape[3] - self.kernel_size) / self.stride + 1
12        self.output = np.zeros([self.input.shape[0], self.input.shape[1], height_out,
13            width_out])
14        for idxn in range(self.input.shape[0]):
15            for idxc in range(self.input.shape[1]):
16                for idxh in range(height_out):
17                    for idxw in range(width_out):
18                        # TODO: 计算最大池化层的前向传播, 取池化窗口内的最大值
19                        self.output[idxn, idxc, idxh, idxw] = _____
20        return self.output

```

图 3.4 最大池化层的实现示例

```

1 # file: layer_2.py
2 class FlattenLayer(object):
3     def __init__(self, input_shape, output_shape): # 层的初始化
4         self.input_shape = input_shape
5         self.output_shape = output_shape
6     def forward(self, input): # 前向传播的计算
7         # matconvnet feature map dim: [N, height, width, channel]
8         # our feature map dim: [N, channel, height, width]
9         self.input = np.transpose(input, [0, 2, 3, 1])
10        self.output = self.input.reshape([self.input.shape[0]] + list(self.output_shape))
11    )
12    return self.output

```

图 3.5 Flatten 层的实现示例

3.1.5.3 网络结构模块

与第2.1节实验类似，本实验的网络结构模块也用一个类来定义 VGG19 神经网络，用类的成员函数来定义 VGG19 的初始化、建立网络结构、神经网络参数初始化等基本操作。VGG19 的网络结构模块的程序示例如图3.6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。为方便起见，本实验在网络初始化时仅设定每层的名称，在建立网络结构时再设定每层的具体超参数。
- 建立网络结构：定义整个神经网络的拓扑结构，设定每层的超参数，实例化基本单元模块中定义的层并将这些层堆叠，组成 VGG19 网络结构。根据表3.1中 VGG19 的网络结构和每层的超参数进行实例化。注意每个卷积层和 3 个全连接层中的前 2 个全连接层后面都跟随有 ReLU 层作为激活函数。此外，pool5 层和 fc6 层中间有一个 flatten 层改变特征图的维度。最后是 Softmax 层计算分类概率。
- 神经网络参数初始化：依次调用神经网络中包含参数的层的参数初始化函数。在本实验中，VGG19 中的 16 个卷积层和 3 个全连接层包含参数，因此需要依次调用其参数初始化函数。

3.1.5.4 网络推断模块

VGG19 的网络推断模块程序示例如图3.7所示。与第2.1节的实验类似，网络推断模块同样包含 VGG19 网络的前向传播、VGG19 网络参数的加载、推断函数主体等操作，这些操作用 VGG19 神经网络类的成员函数来定义：

- 神经网络的前向传播：前向传播的输入是预处理后的图像。首先将预处理后的图像输入到 VGG19 网络的第一层；然后根据之前定义的 VGG19 网络的结构，顺序依次调用每层的前向传播函数，每层的输出作为下一层的输入。由于 VGG19 中的网络层数较多，可以利用网络初始化时定义的层队列，建立循环实现前向传播。
- 神经网络参数的加载：利用官方训练好的 VGG19 模型参数，依次将其中的参数加载到 VGG19 对应的层中。本实验使用的官方模型的下载地址为<http://www.vlfeat.org/matconvnet/models/beta16/imagenet-vgg-verydeep-19.mat>。VGG19 中包含参数的网络层是卷积层和全连接层，可以根据层的编号依次读入对应卷积层和全连接层的权重和偏置。注意在本实验的网络初始化中，在 pool5 层和 fc6 层之间添加了 flatten 层来改变特征图

```

1 # file: vgg_cpu.py
2 class VGG19(object):
3     def __init__(self, param_path='imagenet-vgg-verydeep-19.mat'):
4         # 神经网络的初始化
5         self.param_path = param_path
6         self.param_layer_name = (
7             'conv1_1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1',
8             'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2', 'pool2',
9             'conv3_1', 'relu3_1', 'conv3_2', 'relu3_2', 'conv3_3', 'relu3_3', 'conv3_4', 'relu3_4',
10            , 'pool3',
11            'conv4_1', 'relu4_1', 'conv4_2', 'relu4_2', 'conv4_3', 'relu4_3', 'conv4_4', 'relu4_4',
12            , 'pool4',
13            'conv5_1', 'relu5_1', 'conv5_2', 'relu5_2', 'conv5_3', 'relu5_3', 'conv5_4', 'relu5_4',
14            , 'pool5',
15            'flatten', 'fc6', 'relu6', 'fc7', 'relu7', 'fc8', 'Softmax')
16     def build_model(self): # 建立网络结构
17         # TODO: 定义 VGG19 的网络结构
18         self.layers = {}
19         self.layers['conv1_1'] = ConvolutionalLayer(3, 3, 64, 1, 1)
20         self.layers['relu1_1'] = ReLULayer()
21         self.layers['conv1_2'] = ConvolutionalLayer(3, 64, 64, 1, 1)
22         self.layers['relu1_2'] = ReLULayer()
23         self.layers['pool1'] = MaxPoolingLayer(2, 2)
24
25         self.layers['conv2_1'] = ConvolutionalLayer(3, 64, 64, 1, 1)
26         self.layers['relu2_1'] = ReLULayer()
27         self.layers['conv2_2'] = ConvolutionalLayer(3, 64, 64, 1, 1)
28         self.layers['relu2_2'] = ReLULayer()
29         self.layers['pool2'] = MaxPoolingLayer(2, 2)
30
31         self.layers['conv3_1'] = ConvolutionalLayer(3, 64, 128, 1, 1)
32         self.layers['relu3_1'] = ReLULayer()
33         self.layers['conv3_2'] = ConvolutionalLayer(3, 128, 128, 1, 1)
34         self.layers['relu3_2'] = ReLULayer()
35         self.layers['conv3_3'] = ConvolutionalLayer(3, 128, 128, 1, 1)
36         self.layers['relu3_3'] = ReLULayer()
37         self.layers['conv3_4'] = ConvolutionalLayer(3, 128, 128, 1, 1)
38         self.layers['relu3_4'] = ReLULayer()
39
40         self.layers['pool3'] = MaxPoolingLayer(2, 2)
41
42         self.layers['conv4_1'] = ConvolutionalLayer(3, 128, 256, 1, 1)
43         self.layers['relu4_1'] = ReLULayer()
44         self.layers['conv4_2'] = ConvolutionalLayer(3, 256, 256, 1, 1)
45         self.layers['relu4_2'] = ReLULayer()
46         self.layers['conv4_3'] = ConvolutionalLayer(3, 256, 256, 1, 1)
47         self.layers['relu4_3'] = ReLULayer()
48         self.layers['conv4_4'] = ConvolutionalLayer(3, 256, 256, 1, 1)
49         self.layers['relu4_4'] = ReLULayer()
50
51         self.layers['pool4'] = MaxPoolingLayer(2, 2)
52
53         self.layers['conv5_1'] = ConvolutionalLayer(3, 256, 512, 1, 1)
54         self.layers['relu5_1'] = ReLULayer()
55         self.layers['conv5_2'] = ConvolutionalLayer(3, 512, 512, 1, 1)
56         self.layers['relu5_2'] = ReLULayer()
57         self.layers['conv5_3'] = ConvolutionalLayer(3, 512, 512, 1, 1)
58         self.layers['relu5_3'] = ReLULayer()
59         self.layers['conv5_4'] = ConvolutionalLayer(3, 512, 512, 1, 1)
60         self.layers['relu5_4'] = ReLULayer()
61
62         self.layers['pool5'] = MaxPoolingLayer(2, 2)
63
64         self.layers['flatten'] = FlattenLayer([512, 7, 7], [512*7*7])
65
66         self.layers['fc6'] = FullyConnectedLayer(512*7*7, 4096)
67         self.layers['relu6'] = ReLULayer()
68
69         self.layers['fc7'] = FullyConnectedLayer(4096, 4096)
70         self.layers['relu7'] = ReLULayer()
71
72         self.layers['fc8'] = FullyConnectedLayer(4096, 1000)
73         self.layers['Softmax'] = SoftmaxLossLayer()
74
75         self.update_layer_list = []
76         for layer_name in self.layers.keys():
77             if 'conv' in layer_name or 'fc' in layer_name:
78                 self.update_layer_list.append(layer_name)
79
80     def init_model(self): # 神经网络参数初始化
81         for layer_name in self.update_layer_list:
82             self.layers[layer_name].init_param()

```

图 3.6 VGG19 的网络结构模块实现示例

的维度，而官方提供的模型不包含 flatten 层，因此 fc6 层及之后的层在读取参数时需要偏移。同时值得注意的是，VGG19 官方模型使用的深度学习平台 MatConvNet^[6] 的卷积权重的存储方式与本实验不同。MatConvNet 中卷积权重维度为 $H \times W \times C_{in} \times C_{out}$ ，而本实验中权重的维度为 $C_{in} \times H \times W \times C_{out}$ 。为防止使用官方模型计算出现错误，在读取卷积层权重时需要对输入权重做维度交换，保持与 MatConvNet 的权重存储方式一致。此外还可以从该模型中读取预处理图像时使用的图像均值。

- 神经网络推断函数主体：本实验仅需要对给定的一张图像进行分类，因此给定一张预处理好的图像，执行网络前向传播函数即可获得 VGG19 预测的 1000 个类别的分类概率，然后取其中概率最大的类别作为最终预测的分类类别。在实际应用中，可能需要对一个数据集中的多张测试图像依次进行分类，然后与测试图像对应的标记进行比对，即可得到测试数据集的分类正确率。

```

1 # file: vgg_cpu.py
2 def load_model(self): # 加载神经网络参数
3     params = scipy.io.loadmat(self.param_path)
4     self.image_mean = params['normalization'][0][0][0]
5     self.image_mean = np.mean(self.image_mean, axis=(0, 1))
6     for idx in range(43):
7         if 'conv' in self.param_layer_name[idx]:
8             weight, bias = params['layers'][0][idx][0][0][0][0]
9             # matconvnet: weights dim [height, width, in_channel, out_channel]
10            # ours: weights dim [in_channel, height, width, out_channel]
11            weight = np.transpose(weight,[2,0,1,3])
12            bias = bias.reshape(-1)
13            self.layers[self.param_layer_name[idx]].load_param(weight, bias)
14        if idx >= 37 and 'fc' in self.param_layer_name[idx]:
15            weight, bias = params['layers'][0][idx-1][0][0][0][0]
16            weight = weight.reshape([weight.shape[0]*weight.shape[1]*weight.shape[2],
17                                     weight.shape[3]])
18            self.layers[self.param_layer_name[idx]].load_param(weight, bias)
19
20    def forward(self): # 神经网络的前向传播
21        current = self.input_image
22        for idx in range(len(self.param_layer_name)):
23            current = self.layers[self.param_layer_name[idx]].forward(current)
24        return current
25
26    def evaluate(self): # 推断函数主体
27        prob = self.forward()
28        top1 = np.argmax(prob[0])
29        print('Classification result: id = %d, prob = %f' % (top1, prob[0, top1]))

```

图 3.7 VGG19 的网络推断模块实现示例

3.1.5.5 实验完整流程

完成 VGG19 的每个模块后，就可以用这些模块来实现给定图像的分类。VGG19 进行图像分类的完整流程的程序示例如图3.8所示。首先实例化 VGG19 网络对应的类，建立 VGG19 的网络结构，并对每层的参数进行初始化，然后从官方模型中加载每层的参数，之后加载给定的图像并进行预处理，最后调用网络推断模块获得最终的图像分类结果。

```

1 # file: vgg_cpu.py
2 if __name__ == '__main__':
3     vgg = VGG19()
4     vgg.build_model()
5     vgg.init_model()
6     vgg.load_model()
7     vgg.load_image('cat1.jpg')
8     vgg.evaluate()

```

图 3.8 VGG19 进行图像分类的完整流程实现示例

3.1.5.6 实验运行

根据第3.1.5.1节~第3.1.5.5节的描述补全 layer_1.py、layer_2.py、vgg_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_chap_2_3目录下。

```

1 # 登录云平台
2 ssh root@xxx.xxx.xxx.xxx -pxxxxx
3 # 进入 code_chap_2_3_student 目录
4 cd /opt/code_chap_2_3/code_chap_2_3_student
5 # 初始化环境
6 source env.sh
7

```

2. 代码实现

补全 stu_upload 中的 layer_1.py、layer_2.py、vgg_cpu.py 文件。

```

1 # 进入实验目录
2 cd exp_3_1_vgg
3 # 补全 layers_1.py, layers_2.py, vgg_cpu.py
4 vim stu_upload/layers_1.py
5 vim stu_upload/layers_2.py
6 vim stu_upload/vgg_cpu.py
7

```

3. 运行实验

```

1 # 运行完整实验
2 python main_exp_3_1.py
3

```

3.1.6 实验评估

为验证实验代码的正确性，选择如图3.9所示猫咪的图像进行分类测试。该猫咪图像的真实类别为 tabby cat，对应 ImageNet 数据集类别编号的 281。实验结果将该图像的类别编

号判断为 281。通过查询 ImageNet 数据集类别编号对应的具体类别，编号 281 对应 tabby cat，说明利用 VGG19 网络推断得到了正确的图像类别。



图 3.9 测试猫咪图像示例

本实验的评估标准设定如下：

- 60 分标准：给定卷积层和池化层的前向传播输入矩阵和参数值，可以得到正确的前向传播输出矩阵。
- 80 分标准：建立 VGG19 网络后，给定 VGG19 的网络参数值和输入图像，可以得到正确的 pool5 层输出结果。
- 100 分标准：建立 VGG19 网络后，给定 VGG19 的网络参数值和输入图像，可以得到正确的 Softmax 层输出结果和正确的图像分类结果。

3.1.7 实验思考

- 1) 在实现深度神经网络基本单元时，如何确保一个层的实现是正确的？
- 2) 在实现深度神经网络后，如何确保整个网络的实现是正确的？如果是网络中的某个层计算有误，如何快速定位到有错误的层？
- 3) 如何计算深度神经网络的每层计算量（乘法数量和加法数量）？如何计算整个网络的前向传播时间和网络中每层的前向传播时间？深度神经网络的每层计算量和每层前向传播时间之间有什么关系？

3.2 基于 DLP 平台实现图像分类

3.2.1 实验目的

巩固卷积神经网络的设计原理，能够使用 pycnml 库提供的 Python 接口将 VGG19^[4] 网络模型移植到 DLP 上，实现图像分类。具体包括：

- 1) 使用 pycnml 库实现卷积、池化等基本网络模块。
- 2) 使用提供的 pycnml 库实现 VGG19 网络。
- 3) 分析并比较 DLP 和 CPU 运行 VGG19 进行图像分类的性能。

实验工作量：约 40 行代码，约需 1 个小时。

3.2.2 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：ImageNet。

3.2.3 实验内容

本实验调用 DLP 平台上的 pycnml 库来搭建 VGG19 网络进行图像分类。模块划分方式与第3.1节实验类似，分别为数据加载模块、基本单元模块、网络结构模块和网络推断模块。

3.2.4 实验步骤

3.2.4.1 数据加载模块

数据加载模块实现数据读取和预处理，程序示例如图3.27所示。由于 Python 语言限制，调用 pycnml 库的 Python 接口前需要将数据类型从 numpy.float32 转换为 numpy.float64。

```

1 # file: vgg19_demo.py
2 def load_image(self, image_dir):
3     # 读取图像数据
4     self.image = image_dir
5     image_mean = np.array([123.68, 116.779, 103.939])
6     print('Loading and preprocessing image from ' + image_dir)
7     input_image = scipy.misc.imread(image_dir)
8     input_image = scipy.misc.imresize(input_image,[224,224,3])
9     input_image = np.array(input_image).astype(np.float32)
10    input_image -= image_mean
11    input_image = np.reshape(input_image, [1]+list(input_image.shape))
12    # input dim [N, channel, height, width]
13    input_image = np.transpose(input_image, [0, 3, 1, 2])
14    self.input_data = input_image.flatten().astype(np.float)
15    # 将图片加载到 DLP 上
16    self.net.setInputData(input_data)

```

图 3.10 VGG19 的数据加载模块 DLP 实现示例

3.2.4.2 基本单元模块

VGG19 中包含的卷积层、ReLU 层、最大池化层、全连接层和 Softmax 层可以直接调用 pycnml 库来实现对应层的初始化、参数加载、前向传播等操作。pycnml 的使用方式可以参考第2.2.2.2节的示例。

3.2.4.3 网络结构模块

与第2.2.5.3节类似，网络结构模块也使用一个类来定义 VGG19 网络，可以直接使用 pycnml 封装好的基本模块接口来定义。网络结构模块的程序示例如下面程序所示，其中定义了以下成员函数：

- 神经网络初始化：初始化部分创建 pycnml.CnmlNet() 的实例 net。
- 建立神经网络结构：首先加载数据和权重的量化参数，然后调用 net 中创建网络层的接口定义整个神经网络的拓扑结构，并设定每层的超参数。

```

1 # file: vgg19_demo.py
2 class VGG19(object):
3     def __init__(self):
4         # 初始化网络，创建pycnml.CnmlNet() 实例 net
5         self.net = pycnml.CnmlNet()
6         self.input_quant_params = []
7         self.filter_quant_params = []
8
9     def build_model(self,
10                 param_path='../../data/vgg19_data/imagenet-vgg-verydeep-19.mat',
11                 quant_param_path='../../data/vgg19_data/vgg19_quant_param_new.npz'):
12         self.param_path = param_path
13         # 加载量化参数
14         params = np.load(quant_param_path)
15         input_params = params['input']
16         filter_params = params['filter']
17         for i in range(0, len(input_params), 2):
18             self.input_quant_params.append(pycnml.QuantParam(int(input_params[i]), float(
19             input_params[i+1])))
20             for i in range(0, len(filter_params), 2):
21                 self.filter_quant_params.append(pycnml.QuantParam(int(filter_params[i]), float(
22                 filter_params[i+1])))
23         # TODO: 使用 net 的 createXXXLayer 接口搭建 VGG19 网络
24         self.net.setInputShape(1, 3, 224, 224)
25         # convl_1
26         self.net.createConvLayer('convl_1', 64, 3, 1, 1, 1, self.input_quant_params[0])
27         # relul_1
28         self.net.createReLUlayer('relul_1')
29         # convl_2
30         self.net.createConvLayer('convl_2', 64, 3, 1, 1, 1, self.input_quant_params[1])
31         # relul_2
32         self.net.createReLUlayer('relul_2')
33         # pool1
34         -----
35         -----
36         # fc8
37         self.net.createMlpLayer('fc8', 1000, self.input_quant_params[18])
38         # Softmax
39         self.net.createSoftmaxLayer('Softmax', 1)

```

3.2.4.4 网络推断模块

DLP 实现的 VGG19 的网络推断模块程序示例如下面程序所示。同样划分为参数加载、前向传播、推断函数主体等操作，这些操作使用 VGG19 神经网络类的成员函数来定义：

- 神经网络参数的加载：VGG19 网络参数包括卷积层和全连接层的权重和偏置。首先读取量化过的 VGG19 预训练模型文件，然后循环遍历 net 中的所有层，如果当前层是卷积或全连接层，则将对应的权重、偏置以及量化参数加载到层中。将模型文件读入内存之后，也需要做两方面的处理：一方面，训练得到的模型中权重维度为 $H \times W \times C_{in} \times C_{out}$ ，而 DLP 处理网络层时权重的维度为 $C_{out} \times C_{in} \times H \times W$ ，因此需要对读取的权重做一次维度交换，

使其与 DLP 中权重的维度一致；另一方面，需要手动将 Numpy 数据类型转为 np.float64 类型。

- 神经网络的前向传播：将经过预处理的图像输入，net.forward 函数会自动遍历调用 net 中的每一层的前向传播函数，并返回最后一层的结果。

- 神经网络推断函数主体：与第3.1.5.4节的 CPU 实现类似，给定一张经过预处理的图像数据，执行网络的前向传播函数即可得到 VGG19 预测的 1000 个类别的分类概率，然后选取概率最高的类别作为网络最终预测的分类类别。

```

1 # file: vgg19_demo.py
2 def load_model(self): # 加载神经网络参数
3     print('Loading parameters from file ' + self.param_path)
4     params = scipy.io.loadmat(self.param_path)
5     self.image_mean = params['normalization'][0][0][0]
6     self.image_mean = np.mean(self.image_mean, axis=(0, 1))
7     count = 0
8     for idx in range(self.net.size()):
9         if 'conv' in self.net.getLayerName(idx):
10             weight, bias = params['layers'][0][idx][0][0][0][0]
11             # matconvnet: weights dim [height, width, in_channel, out_channel]
12             # ours: weights dim [out_channel, in_channel, height, width]
13             weight = np.transpose(weight,[3,2,0,1]).flatten().astype(np.float)
14             bias = bias.reshape(-1).astype(np.float)
15             self.net.loadParams(idx, weight, bias, self.filter_quant_params[count])
16             count += 1
17         if 'fc' in self.net.getLayerName(idx):
18             weight, bias = params['layers'][0][idx-1][0][0][0][0]
19             weight = weight.reshape([weight.shape[0]*weight.shape[1]*weight.shape[2], weight.shape
20             [3]])
21             weight = np.transpose(weight, [1, 0]).flatten().astype(np.float)
22             bias = bias.reshape(-1).astype(np.float)
23             self.net.loadParams(idx, weight, bias, self.filter_quant_params[count])
24             count += 1
25
26     def forward(self): # 神经网络的前向传播
27         return self.net.forward()
28
29     def get_top5(self, label):
30         # 打印推理的时间
31         start = time.time()
32         self.forward()
33         end = time.time()
34         print('inference time: %f'%(end - start))
35         result = self.net.getOutputData()
36         # 打印 top1/5 结果
37         top1 = False
38         top5 = False
39         print('----- Top 5 of ' + self.image + ' -----')
40         prob = sorted(list(result), reverse=True)[:6]
41         if result.index(prob[0]) == label:
42             top1 = True
43         for i in range(5):
44             top = prob[i]
45             idx = result.index(top)
46             if idx == label:
47                 top5 = True
48             print('%f - %top + self.labels[idx].strip())
49         return top1, top5
50
51     def evaluate(self, file_list): # 推断函数主体

```

```

51 top1_num = 0
52 top5_num = 0
53 total_num = 0
54 # 读取标签
55 self.labels = []
56 with open('synset_words.txt', 'r') as f:
57     self.labels = f.readlines()
58 # 记录推断所有图片的总时间
59 start = time.time()
60 with open(file_list, 'r') as f:
61     file_list = f.readlines()
62 total_num = len(file_list)
63 for line in file_list:
64     image = line.split()[0].strip()
65     label = int(line.split()[1].strip())
66     self.load_image(image)
67     top1,top5 = self.get_top5(label) # 获取推断结果
68     if top1:
69         top1_num += 1
70     if top5:
71         top5_num += 1
72 end = time.time()
73 print('Global accuracy : ')
74 print('accuracy1: %f (%d/%d)'% (float(top1_num)/float(total_num), top1_num, total_num))
75 print('accuracy5: %f (%d/%d)'% (float(top5_num)/float(total_num), top5_num, total_num))
76 print('Total execution time: %f'%(end - start))

```

3.2.4.5 实验完整流程

完成以上所有模块后，就可以调用上述模块中的函数，在 DLP 上运行 VGG19 网络实现给定图像的分类。完整的流程程序示例如图3.11所示。与第3.1节的 CPU 实现类似，首先实例化 VGG19 网络的类，其次建立网络结构，设置每层的超参数，然后读取模型文件为每层加载参数，最后输入待分类的图像并调用推断模块获得网络的预测结果。

```

1 # file: vgg19_demo.py
2 if __name__ == '__main__':
3     vgg = VGG19()
4     vgg.build_model()
5     vgg.load_model()
6     vgg.evaluate('file_list')

```

图 3.11 VGG19 进行图像分类的完整流程 DLP 实现示例

3.2.4.6 实验运行

根据第3.2.4.1节~第3.2.4.5节的描述补全 vgg19_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 目录下。

```

1 # 登录云平台
2 ssh root@xxx.xxx.xxx.xxx -p xxxxx
3 # 进入 code_chap_2_3_student 目录
4 cd /opt/code_chap_2_3/code_chap_2_3_student
5 # 初始化环境
6 source env.sh
7

```

2. 代码实现

补全 stu_upload 中的 vgg19_demo.py 文件。

```

1 # 进入实验目录
2 cd exp_3_1_vgg
3 # 补全 vgg19_demo.py
4 vim stu_upload/vgg19_demo.py
5

```

3. 运行实验

```

1 # 运行完整实验
2 python main_exp_3_2.py
3

```

3.2.5 实验评估

本实验仍然选择图3.9所示猫咪的图像进行分类测试，该猫咪图像的真实类别为 tabby cat，对应 ImageNet 数据集类别编号的 281。若实验结果将该图像的类别编号判断为 281，则可认为判断正确。性能评判标准为预测猫咪类别时 VGG19 网络 forward 函数运行的时间。

本实验的评分标准设定如下：

- 100 分标准：使用 pycnml 搭建 VGG19 网络，给定 VGG19 的网络参数值和输入图像，可以得到正确的 Softmax 层输出结果和正确的图像分类结果。

3.2.6 实验思考

1) 阅读 pycnml/src/net.cpp 中 forward 函数的实现，比较 DLP 在计算哪些层时比 CPU 要快，为什么？

2) 观察 forward 函数的实现，在 VGG19 网络的一次完整推断过程中，DLP 每执行完一层都需要和 CPU 交互一次，这种交互是否有必要？有什么办法可以避免这种交互吗？

3.3 非实时图像风格迁移

3.3.1 实验目的

掌握深度学习的训练方法，能够使用 Python 语言基于 VGG19 网络模型实现非实时图像风格迁移^[7]。具体包括：

- 1) 加深对卷积神经网络的理解，利用 VGG19 模型进行图像特征提取。
- 2) 使用 Python 语言实现风格迁移中风格和内容损失函数的计算，加深对非实时风格迁移的理解。
- 3) 使用 Python 语言实现非实时风格迁移中迭代求解风格化图像的完整流程，为后续实现实时风格迁移并建立更复杂的综合实验奠定基础。

实验工作量：约 20 行代码，约需 3 个小时。

3.3.2 背景介绍

图像风格迁移根据给定的目标风格图像和目标内容图像求解风格迁移图像，使风格迁移图像在风格上与目标风格图像一致，在内容上与目标内容图像一致。图像风格迁移分为非实时风格迁移与实时风格迁移。非实时风格迁移仅对当前给定的目标内容图像进行风格化，实现较为简单，但需要对每张输入的内容图像做训练。而实时风格迁移训练一个模型，对任意内容图像均可以生成风格化图像，实现相对复杂。

风格迁移通常用 VGG 模型（如 VGG19）提取图像的特征，然后计算风格迁移图像与目标风格（内容）图像的风格（内容）损失作为风格（内容）损失函数。在非实时风格迁移中，计算风格（内容）损失并进行反向传播，获得风格迁移图像的梯度，更新风格迁移图像。通过多次迭代，不断减小风格迁移图像与目标风格（内容）图像的风格（内容）损失，最终获得风格化后的图像。在非实时风格迁移中，通常用加入噪声的目标内容图像作为初始的风格迁移图像。完整的非实时风格迁移过程见图3.12。下面详细介绍非实时风格迁移中使用的内容损失函数和风格损失函数。

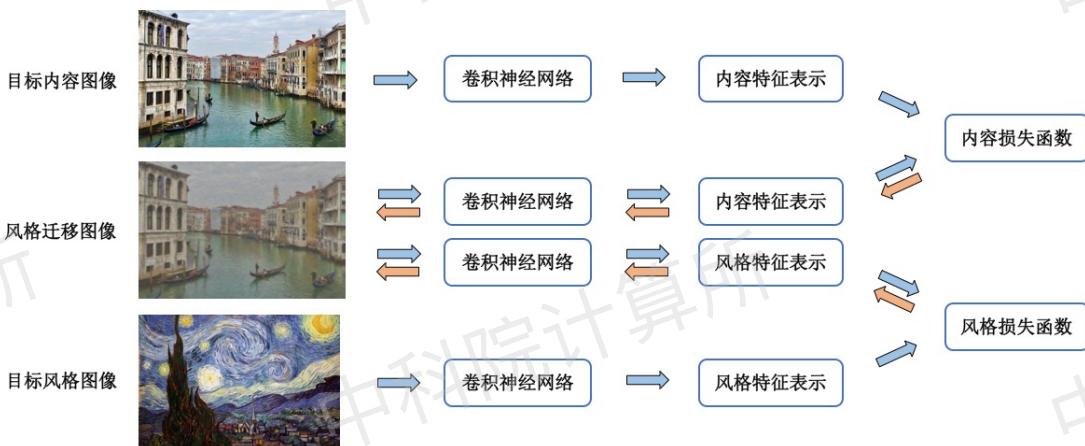


图 3.12 非实时风格迁移

内容损失函数

内容损失层用于计算风格迁移图像的第 l 层特征图与目标内容图像的第 l 层特征图的内容损失。假设风格迁移图像的第 l 层特征图为 \mathbf{X}^l ，是维度为 $N \times C \times H \times W$ 的四维矩阵， N 、 C 、 H 、 W 分别代表输入特征图的样本个数（在本实验中 $N = 1$ ）、通道数、高和宽。假设目标内容图像的第 l 层特征图为 \mathbf{Y}^l ，维度同样为 $N \times C \times H \times W$ ，则目标内容图像的该层特征图可视为是内容损失层的标记。内容损失 L 用 \mathbf{X}^l 和 \mathbf{Y}^l 之间的欧式距离表示，计算公式为：

式为

$$L_{content} = \frac{1}{2NCHW} \sum_{n,c,h,w} (\mathbf{X}^l(n, c, h, w) - \mathbf{Y}^l(n, c, h, w))^2 \quad (3.10)$$

其中 $n \in [1, N]$ 、 $c \in [1, C]$ 、 $h \in [1, H]$ 、 $w \in [1, W]$ 均为整数，用于表示特征图上的位置， $\mathbf{X}^l(n, c, h, w)$ 和 $\mathbf{Y}^l(n, c, h, w)$ 分别表示风格迁移图像和目标内容图像的第 l 层特征图上第 n 个样本第 c 通道高为 h 宽为 w 位置处的特征值。内容损失为特征图上每个位置的平均欧式距离，因此需要求和后除以特征图中特征值的数量。

反向传播计算时，根据内容损失的计算公式(3.10)可计算出内容损失对于风格迁移图像的特征图 \mathbf{X}^l 的梯度 $\nabla_{\mathbf{X}^l} L_{content}$ 为

$$\nabla_{\mathbf{X}^l} L_{content}(n, c, h, w) = \frac{1}{NCHW} (\mathbf{X}^l(n, c, h, w) - \mathbf{Y}^l(n, c, h, w)) \quad (3.11)$$

本实验用 conv4_2 之后的 ReLU 层（即 relu4_2）的输出特征图来计算内容损失函数。因此公式(3.11)与《智能计算系统》教材第 3.6.1 节中的内容损失梯度略微不同。教材的公式中，当风格迁移图像某位置的值小于 0 时，对应位置的内容损失梯度置为 0。这是由于课程教材中使用的是风格内容图像在 conv4_2 卷积层的特征，而本实验中使用的是卷积之后的 ReLU 层的特征图，经过 ReLU 层的反向传播后，本实验计算的损失与教材的公式结果相同。

风格损失函数

风格损失层用于计算风格迁移图像的第 l 层特征图与目标风格图像的第 l 层特征图的风格损失。假设风格迁移图像的第 l 层特征图为 \mathbf{X}^l ，是维度为 $N \times C \times H \times W$ 的四维矩阵， N, C, H, W 分别代表输入特征图的样本个数（在本实验中 $N = 1$ ）、通道数、高和宽。假设目标风格图像的该层特征图为 \mathbf{Y}^l ，维度同样为 $N \times C \times H \times W$ 。在前向传播的计算过程中，首先利用 Gram 矩^[7] 计算风格迁移图像和目标风格图像的风格特征 \mathbf{G} 和 \mathbf{A} ，用第 i 和 j 通道的特征图内积表示，计算公式为

$$\begin{aligned} \mathbf{G}^l(n, i, j) &= \sum_{h,w} \mathbf{X}^l(n, i, h, w) \mathbf{X}^l(n, j, h, w) \\ \mathbf{A}^l(n, i, j) &= \sum_{h,w} \mathbf{Y}^l(n, i, h, w) \mathbf{Y}^l(n, j, h, w) \end{aligned} \quad (3.12)$$

其中 $n \in [1, N]$ ， $i, j \in [1, C]$ 代表第 i 和 j 通道的特征图， $h \in [1, H]$ 和 $w \in [1, W]$ 代表特征图上的水平和垂直位置，风格特征 \mathbf{G}^l 和 \mathbf{A}^l 的维度均为 $N \times C \times C$ 。第 l 层的风格损失 L_{style}^l 为

$$L_{style}^l = \frac{1}{4NC^2H^2W^2} \sum_{n,i,j} (\mathbf{G}^l(n, i, j) - \mathbf{A}^l(n, i, j))^2 \quad (3.13)$$

风格损失函数为各层风格损失之和：

$$L_{style} = \sum_l \omega_l L_{style}^l \quad (3.14)$$

其中， ω_l 是计算风格损失时第 l 层损失的权重。

计算反向传播时，根据风格损失的计算公式(3.12)和(3.14)，可得第 l 层风格损失对风格

迁移图像特征图 \mathbf{X}^l 的梯度^①:

$$\nabla_{\mathbf{X}^l} L_{style}^l(n, i, h, w) = \frac{1}{NC^2H^2W^2} \sum_j \mathbf{X}^l(n, j, h, w)(\mathbf{G}^l(n, j, i) - \mathbf{A}^l(n, j, i)) \quad (3.15)$$

其中 $n \in [1, N]$, $i, j \in [1, C]$ 代表特征图的第 i 和 j 通道, $h \in [1, H]$ 和 $w \in [1, W]$ 代表特征图上的水平和垂直位置。

损失函数

风格迁移的损失函数为内容损失和风格损失的加权和:

$$L_{total} = \alpha L_{content} + \beta L_{style} \quad (3.16)$$

其中, α 和 β 为权重。

Adam 优化器

训练神经网络时, 一般使用批量随机梯度下降算法对网络参数进行更新。批量随机梯度下降算法对网络的所有参数使用相同的学习率, 且在无人工更改的情况下会保持学习率固定不变。而在非实时风格迁移中, 使用 Adam 算法^[8] 对风格迁移图像进行更新。相对于批量随机梯度下降算法, Adam 算法利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率, 因此收敛速度更快, 训练过程也更加平稳。给定待更新的风格迁移图像 \mathbf{X} 和梯度 $\nabla_{\mathbf{X}} L$, 当前迭代次数 t , 设定 Adam 优化器的超参数 $\beta_1 = 0.9$ 、 $\beta_2 = 0.999$ 、 $\epsilon = 10^{-8}$, 初始学习率 η , 则风格迁移图像的更新公式为:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\mathbf{X}} L \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) \nabla_{\mathbf{X}} L^2 \\ \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \\ \mathbf{X} &\leftarrow \mathbf{X} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{aligned} \quad (3.17)$$

其中 m_t 是梯度的一阶矩估计, v_t 是梯度的二阶矩估计, \hat{m}_t 和 \hat{v}_t 表示 m_t 和 v_t 无偏校正后的结果。

更多关于非实时风格迁移的介绍详见《智能计算系统》课程教材第 3.6.1 节。

3.3.3 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)。

^①此公式的情况与公式(3.11)类似, 由于课程教材中使用的是卷积层的特征图, 而本实验中使用的是 ReLU 层的特征图, 因此课程教材的公式中当风格迁移图像某位置的值小于 0 时对应位置的风格损失梯度置为 0。经过 ReLU 层的反向传播后, 本实验计算的损失将与课程教材的公式结果相同。

3.3.4 实验内容

本实验利用 VGG19 网络实现非实时风格迁移。非实时风格迁移仅需计算当前给定的内容图像的风格化图像，与实时风格迁移相比实现较为简单。为确保生成的风格迁移图像与目标风格（内容）图像的一致性，首先利用 VGG19 模型提取内容图像和风格图像的特征，随后计算风格迁移图像与目标风格（内容）图像的风格（内容）损失，然后对损失进行反向传播，更新生成图像。通过多次迭代，不断减小生成图像与目标风格（内容）图像的风格（内容）损失，最终获得风格化后的图像。在非实时风格迁移实验中，仅用到 VGG19 模型的卷积层和池化层，不用全连接层和 Softmax 层。

工程实现时，依然按照第3.1节实验的模块划分方法，并基于第3.1实验中已实现的模块进行改进。由于非实时风格迁移中使用 VGG19 网络进行特征提取，本实验可以利用第3.1节实验中的部分模块。由于本实验只涉及风格化图像的迭代求解过程，因此本实验仅包括数据加载模块、基本单元模块、网络结构模块、网络训练模块，不包括网络推断模块。

3.3.5 实验步骤

3.3.5.1 数据加载模块

非实时风格迁移需要能够读入内容图像和风格图像，生成风格图像的初始化图像，在训练过程中保存图像，因此需要实现数据加载、生成图像初始化、图像保存函数，程序示例如图 3.13 所示。具体函数功能为：

- 数据加载：与第3.1节实验类似，实现从文件中读取图像、缩放图像、图像标准化（减去图像均值）、转换图像矩阵维度等过程。需要注意的是，本实验仅使用 VGG19 模型的卷积层和最大池化层进行特征的提取，VGG19 中的全连接层并没有参与计算，而卷积层和最大池化层的输入特征图的高和宽是可以变化的。因此，可以在数据加载模块中将预处理后图像的分辨率作为超参数，方便灵活调整风格迁移图像的分辨率。
- 生成图像初始化：在目标内容图像中加入随机高斯噪声来初始化风格迁移图像。
- 图像保存：保存最终获得的风格迁移图像。其过程与图像的读取和预处理过程相反，包括转换图像矩阵维度，加上图像均值，缩放图像和保存图像到文件中。

3.3.5.2 基本单元模块

本实验涉及的基本单元模块包括：卷积层、最大池化层、内容损失、风格损失、Adam 优化器。

卷积层

第3.1节实验中已经实现了卷积层的初始化、参数的初始化和加载、前向传播计算等步骤。本实验还需实现卷积层的反向传播计算，用于计算风格迁移图像的梯度。卷积层反向传播的程序示例如图 3.14 所示。

- 反向传播计算：根据公式(3.5)和(3.6)，可以进行卷积层反向传播的计算。首先根据公式(3.5)计算权重和偏置的梯度 $\nabla_w L$ 、 $\nabla_b L$ 以及损失函数对边界扩充后的输入的偏导 $\nabla_{x_{pad}} L$ 。与前向传播过程类似，在工程实现中可以通过四重循环依次计算 $\nabla_w L$ 、 $\nabla_b L$ 、 $\nabla_{x_{pad}} L$ 每个位置的值。之后根据公式(3.6)将 $\nabla_{x_{pad}} L$ 中扩充的边缘裁剪掉即可得到 $\nabla_x L$ 。

```

1 # file: exp_3_3_style_transfer.py
2 def load_image(self, image_dir, image_height, image_width):
3     # 数据加载模块
4     self.input_image = scipy.misc.imread(image_dir)
5     image_shape = self.input_image.shape
6     self.input_image = scipy.misc.imresize(self.input_image, [image_height, image_width
7         , 3])
8     self.input_image = np.array(self.input_image).astype(np.float32)
9     self.input_image -= self.image_mean
10    self.input_image = np.reshape(self.input_image, [1]+list(self.input_image.shape))
11    # input dim [N, channel, height, width]
12    self.input_image = np.transpose(self.input_image, [0, 3, 1, 2])
13    return self.input_image, image_shape
14 def get_random_img(content_image, noise):
15    # 生成风格迁移初始化图像
16    noise_image = np.random.uniform(-20, 20, content_image.shape)
17    random_img = noise_image * noise + content_image * (1 - noise)
18    return random_img
19 def save_image(self, input_image, image_shape, image_dir):
20    # 保存图像
21    input_image = np.transpose(input_image, [0, 2, 3, 1])
22    input_image = input_image[0] + self.image_mean
23    input_image = np.clip(input_image, 0, 255).astype(np.uint8)
24    input_image = scipy.misc.imresize(input_image, image_shape)
25    scipy.misc.imsave(image_dir, input_image)

```

图 3.13 非实时风格迁移的数据加载模块实现示例

```

1 # file: layer_2.py
2 def backward(self, top_diff): # 卷积层的反向传播
3     self.d_weight = np.zeros(self.weight.shape)
4     self.d_bias = np.zeros(self.bias.shape)
5     bottom_diff = np.zeros(self.input_pad.shape)
6     for idxn in range(top_diff.shape[0]):
7         for idxc in range(top_diff.shape[1]):
8             for idxh in range(top_diff.shape[2]):
9                 for idxw in range(top_diff.shape[3]):
10                     # TODO: 计算卷积层的反向传播, 权重、偏置的梯度和本层损失
11                     self.d_weight[:, :, :, idxc] += _____
12                     self.d_bias[idxc] += _____
13                     bottom_diff[idxn, :, idxh*self.stride:idxh*self.stride+self.kernel_size, idxw*self
14                     .stride:idxw*self.stride+self.kernel_size] += _____
15     bottom_diff = bottom_diff[:, :, self.padding:self.padding+self.input.shape[2], self.padding:
16     self.padding+self.input.shape[3]]
17     return bottom_diff

```

图 3.14 卷积层反向传播的实现示例

最大池化层

在第3.1节实验中已经实现了最大池化层前向传播的计算。本实验还需实现最大池化层反向传播的计算，用于计算风格迁移图像的梯度。最大池化层反向传播的程序示例如图 3.15所示。

- 反向传播计算：根据公式(3.8)和(3.9)，可以进行最大池化层反向传播的计算。在反向传播时，仅将后一层损失中对应该池化窗口的值传递给池化窗口内最大值所在位置，其他

位置值置为0。在反向传播时需先根据公式(3.8)计算最大值所在位置，公式(3.8)中 F 代表取最大值所在位置的函数，返回最大值在池化窗口中的坐标向量。在Python中 F 函数可使用`argmax`函数加`unravel_index`函数实现。根据公式(3.9)，利用最大值所在位置可以计算得到最大池化层的损失。

```

1 # file: layer_2.py
2 def backward(self, top_diff): # 最大池化层的反向传播
3     bottom_diff = np.zeros(self.input.shape)
4     for idxn in range(top_diff.shape[0]):
5         for idxc in range(top_diff.shape[1]):
6             for idxh in range(top_diff.shape[2]):
7                 for idxw in range(top_diff.shape[3]):
8                     # TODO: 最大池化层的反向传播，计算池化窗口中最大值位置，并传递损失
9                     max_index = _____
10                    bottom_diff[idxn, idxc, idxh*self.stride+max_index[0], idxw*self.
11 stride+max_index[1]] = _____
12    return bottom_diff

```

图 3.15 池化层反向传播的实现示例

内容损失

内容损失的计算需要在完成内容图像和生成图像的前向传播计算之后，计算生成图像的特征图与内容图像的特征图之间的内容损失；然后，在反向传播计算时，计算内容损失对生成图像的特征图的梯度。具体实现时，内容损失的计算用一个类来定义，前向传播计算和反向传播计算用类成员函数来定义，程序示例如图 3.16 所示。

- 前向传播计算：计算生成图像的某层特征图与内容图像的该层特征图的内容损失。内容损失用风格迁移图像的某层特征图与目标内容图像的该层特征图的欧式距离表示，根据公式(3.10)进行计算。
- 反向传播计算：根据内容损失的反向传播计算公式(3.11)，可计算得到内容损失对于风格迁移图像的特征图的梯度。

```

1 # file: layer_3.py
2 class ContentLossLayer(object):
3     def forward(self, input_layer, content_layer): # 前向传播的计算
4         # TODO: 计算风格迁移图像和目标内容图像的内容损失
5         loss = _____
6         return loss
7     def backward(self, input_layer, content_layer): # 反向传播的计算
8         # TODO: 计算内容损失的反向传播
9         bottom_diff = _____
10        return bottom_diff

```

图 3.16 内容损失计算的实现示例

风格损失

风格损失的计算需要在完成风格图像和生成图像的前向传播计算之后，计算生成图像的特征图与风格图像的特征图之间的风格损失；然后，在反向传播计算时，计算风格损失对生成图像的特征图的梯度。具体实现时，风格损失的计算用一个类来定义，前向传播计算和反向传播计算用类成员函数来定义，程序示例如图 3.17 所示。

- 前向传播的计算：计算生成图像的某层特征图与目标风格图像的该层特征图的风格损失。首先根据公式(3.12)，利用 Gram 矩阵计算风格迁移图像和目标风格图像的风格特征 G 和 A 。然后根据公式(3.14)计算风格损失。

- 反向传播的计算：风格损失对于生成图像的特征图的梯度可根据公式(3.15)计算。

```

1 # file: layer_3.py
2 class StyleLossLayer(object):
3     def forward(self, input_layer, style_layer): # 前向传播的计算
4         # TODO: 计算风格迁移图像和目标风格图像的 Gram 矩阵
5         style_layer_reshape = np.reshape(style_layer, [style_layer.shape[0], style_layer.
6             .shape[1], -1])
7         self.gram_style = _____
8         self.input_layer_reshape = np.reshape(input_layer, [input_layer.shape[0],
9             input_layer.shape[1], -1])
10        self.gram_input = np.zeros([input_layer.shape[0], input_layer.shape[1],
11            input_layer.shape[1]])
12        for idxn in range(input_layer.shape[0]):
13            self.gram_input[idxn, :, :] = _____
14        # TODO: 计算风格迁移图像和目标风格图像的风格损失
15        loss = _____
16        return loss
17    def backward(self, input_layer, style_layer): # 反向传播的计算
18        bottom_diff = np.zeros([input_layer.shape[0], input_layer.shape[1], input_layer.
19            shape[2]*input_layer.shape[3]])
20        for idxn in range(input_layer.shape[0]):
21            # TODO: 计算风格损失的反向传播
22            bottom_diff[idxn, :, :] = _____
23        bottom_diff = np.reshape(bottom_diff, input_layer.shape)
24        return bottom_diff

```

图 3.17 风格损失层的实现示例

Adam 优化器

在非实时风格迁移中，使用 Adam 算法对生成图像进行更新。在实现 Adam 优化器时，首先在初始化函数中设定 Adam 优化器的超参数，如 $\beta_1 = 0.9$ 、 $\beta_2 = 0.999$ 、 $\epsilon = 10^{-8}$ ，初始学习率 η 。然后定义 Adam 优化器中的更新函数，给定待更新的生成图像 X 和梯度 $\nabla_X L$ ，当前迭代次数 t ，根据公式(3.17)计算梯度的一阶矩和二阶矩，分别进行无偏矫正后，对生成图像进行更新。

Adam 优化器的程序示例如图 3.18 所示。

3.3.5.3 网络结构模块

非实时风格迁移也使用 VGG19 网络，因此本实验的网络结构模块与第3.1节实验的网络结构模块基本一致。本实验 VGG19 的网络结构模块程序示例如图3.20所示，网络结构模块中同样包含 VGG19 的初始化、建立网络结构、神经网络的参数初始化等基本操作。其中主要区别在于，本实验仅使用 VGG19 的卷积层和池化层，因此 pool5 层后面的全连接层等部分可以省略。

```

1 # file: exp_3_3_style_transfer.py
2 class AdamOptimizer(object):
3     def __init__(self, lr, diff_shape): # Adam优化器的初始化
4         self.betalpha = 0.9
5         self.betabeta = 0.999
6         self.eps = 1e-8
7         self.lr = lr
8         self.mt = np.zeros(diff_shape)
9         self.vt = np.zeros(diff_shape)
10        self.step = 0
11    def update(self, input, grad): # 参数更新过程
12        self.step += 1
13        self.mt = self.betalpha * self.mt + (1 - self.betalpha) * grad
14        self.vt = self.betabeta * self.vt + (1 - self.betabeta) * np.square(grad)
15        mt_hat = self.mt / (1 - self.betalpha ** self.step)
16        vt_hat = self.vt / (1 - self.betabeta ** self.step)
17        # TODO: 利用梯度的一阶矩和二阶矩的无偏估计更新风格迁移图像
18        output = -----
19    return output

```

图 3.18 Adam 优化器的实现示例

```

1 # file: exp_3_3_style_transfer.py
2 class VGG19(object):
3     def __init__(self, param_path='imagenet-vgg-verydeep-19.mat'):
4         # 神经网络的初始化
5         self.param_path = param_path
6         self.param_layer_name = (
7             'conv1_1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1',
8             'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2', 'pool2',
9             'conv3_1', 'relu3_1', 'conv3_2', 'relu3_2', 'conv3_3', 'relu3_3', 'conv3_4',
10            'relu3_4', 'pool3',
11            'conv4_1', 'relu4_1', 'conv4_2', 'relu4_2', 'conv4_3', 'relu4_3', 'conv4_4',
12            'relu4_4', 'pool4',
13            'conv5_1', 'relu5_1', 'conv5_2', 'relu5_2', 'conv5_3', 'relu5_3', 'conv5_4',
14            'relu5_4', 'pool5')
15     def build_model(self): # 建立网络结构
16         # TODO: 建立 VGG19 网络结构
17         self.layers = {}
18         self.layers['conv1_1'] = ConvolutionalLayer(3, 3, 64, 1, 1)
19         self.layers['relu1_1'] = ReLULayer()
20         self.layers['conv1_2'] = ConvolutionalLayer(3, 64, 64, 1, 1)
21         self.layers['relu1_2'] = ReLULayer()
22         self.layers['pool1'] = MaxPoolingLayer(2, 2)
23
24         self.layers['conv2_1'] = ConvolutionalLayer(3, 64, 128, 1, 1)
25         self.layers['relu2_1'] = ReLULayer()
26         self.layers['conv2_2'] = ConvolutionalLayer(3, 128, 128, 1, 1)
27         self.layers['relu2_2'] = ReLULayer()
28         self.layers['pool2'] = MaxPoolingLayer(2, 2)
29
30         self.layers['conv3_1'] = ConvolutionalLayer(3, 128, 256, 1, 1)
31         self.layers['relu3_1'] = ReLULayer()
32         self.layers['conv3_2'] = ConvolutionalLayer(3, 256, 256, 1, 1)
33         self.layers['relu3_2'] = ReLULayer()
34         self.layers['conv3_3'] = ConvolutionalLayer(3, 256, 256, 1, 1)
35         self.layers['relu3_3'] = ReLULayer()
36         self.layers['conv3_4'] = ConvolutionalLayer(3, 256, 256, 1, 1)
37         self.layers['relu3_4'] = ReLULayer()
38         self.layers['pool3'] = MaxPoolingLayer(2, 2)
39
40         self.layers['conv4_1'] = ConvolutionalLayer(3, 256, 512, 1, 1)
41         self.layers['relu4_1'] = ReLULayer()
42         self.layers['conv4_2'] = ConvolutionalLayer(3, 512, 512, 1, 1)
43         self.layers['relu4_2'] = ReLULayer()
44         self.layers['conv4_3'] = ConvolutionalLayer(3, 512, 512, 1, 1)
45         self.layers['relu4_3'] = ReLULayer()
46         self.layers['conv4_4'] = ConvolutionalLayer(3, 512, 512, 1, 1)
47         self.layers['relu4_4'] = ReLULayer()
48         self.layers['pool4'] = MaxPoolingLayer(2, 2)
49
50         self.layers['conv5_1'] = ConvolutionalLayer(3, 512, 512, 1, 1)
51         self.layers['relu5_1'] = ReLULayer()
52         self.layers['conv5_2'] = ConvolutionalLayer(3, 512, 512, 1, 1)
53         self.layers['relu5_2'] = ReLULayer()
54         self.layers['conv5_3'] = ConvolutionalLayer(3, 512, 512, 1, 1)
55         self.layers['relu5_3'] = ReLULayer()
56         self.layers['conv5_4'] = ConvolutionalLayer(3, 512, 512, 1, 1)
57         self.layers['relu5_4'] = ReLULayer()
58         self.layers['pool5'] = MaxPoolingLayer(2, 2)
59
60         self.update_layer_list = []
61         for layer_name in self.layers.keys():
62             if 'conv' in layer_name or 'fc' in layer_name:
63                 self.update_layer_list.append(layer_name)
64
65     def init_model(self): # 神经网络参数初始化
66         for layer_name in self.update_layer_list:
67             self.layers[layer_name].init_param()

```

图 3.19 非实时风格迁移中 VGG19 的网络结构模块实现示例

3.3.5.4 网络训练模块

本实验通过网络训练模块来迭代求解风格迁移图像。每次迭代过程中，首先做前向传播并计算损失函数，再做反向传播计算风格迁移图像的梯度，然后进行更新。与第2.1节实验中的网络训练模块类似，本实验中的网络训练模块包括训练函数主体、神经网络前向传播、神经网络反向传播等基本步骤。同时由于非实时风格迁移中不需要网络推断模块，因此将神经网络的参数加载步骤也放在网络训练模块中。本实验中 VGG19 的网络训练模块程序示例如图3.20所示。

- 神经网络的前向传播：通常的神经网络前向传播过程如第2.1节实验中所介绍的，将预处理后的图像输入到神经网络的第一层中，再根据之前定义的网络结构顺序依次调用每层的前向传播函数，然后将每层的输出作为下一层的输入，直到得到最后一层的输出结果。但在非实时风格迁移中，计算内容损失和风格损失函数时，可能会用到中间层的特征图，因此本实验中的前向传播函数会将计算内容/风格损失需要使用的层作为输入参数，利用一个字典记录所有这些层的输出结果。
- 神经网络的反向传播：通常的神经网络反向传播过程是利用神经网络最后一层的输出与标记计算损失，之后利用链式法则逆序逐层计算损失函数对每层输入及参数的偏导（损失及参数梯度），最后得到神经网络所有层的参数梯度，如第2.1节实验。在非实时风格迁移的反向传播过程与通常的神经网络存在两方面的区别：一方面，非实时风格迁移在反向传播时不需要计算每层的参数梯度，仅需计算每层的损失，用最终得到的第一层的损失作为风格迁移图像的梯度对其进行更新。另一方面，计算内容损失函数和风格损失函数时需要对多个中间层的特征图计算损失函数，而不一定只对最后一层特征图计算损失函数。因此在实现反向传播时，首先定位所有计算损失的中间层位置，然后以该层开始逆序计算前面每一层的损失，最终得到第一层的损失。
- 加载神经网络参数：此过程与第3.1节实验中加载 VGG19 官方模型参数的过程基本一致。不同之处在于仅需加载卷积层的参数和预处理图像时使用的图像均值，可以省略全连接层的参数。
- 神经网络训练函数主体：由于非实时风格迁移是对输入的风格迁移图像进行更新，而不是对神经网络参数进行更新。为方便实现，将训练函数主体放在下一小节“实验完整流程”中。

3.3.5.5 实验完整流程

完成非实时风格迁移的每个模块之后，就可以利用这些模块进行风格迁移图像的计算。非实时风格迁移的完整流程如图3.21所示。

- 首先确定超参数，包括计算内容损失和风格损失函数时使用 VGG19 的哪些层、数据预加载模块相关的图像缩放分辨率、训练有关的学习率大小、迭代次数、损失函数权重系数等。在本实验中，计算内容损失函数使用的内容损失层为 relu4_2 层，计算风格损失函数使用的风格损失层为 relu1_1、relu2_1、relu3_1、relu4_1、relu5_1 层。
- 其次建立 VGG19 的网络结构并从官方模型中加载参数，同时实例化非实时风格迁移中的内容损失计算、风格损失计算和 Adam 优化器。之后读取给定的内容图像和风格图像

```

1 # file: exp_3_3_style_transfer.py
2 def forward(self, input_image, layer_list): # 前向传播的计算
3     current = input_image
4     layer_forward = {}
5     for idx in range(len(self.param_layer_name)):
6         # TODO: 计算VGG19网络的前向传播
7         current = _____
8         if self.param_layer_name[idx] in layer_list:
9             layer_forward[self.param_layer_name[idx]] = current
10    return layer_forward
11
12 def backward(self, dloss, layer_name): # 反向传播的计算
13     layer_idx = list.index(self.param_layer_name, layer_name)
14     for idx in range(layer_idx, -1, -1):
15         # TODO: 计算VGG19网络的反向传播
16         dloss = _____
17     return dloss

```

图 3.20 非实时风格迁移的网络训练模块实现示例

进行预处理，并计算相应的特征图作为计算内容损失和风格损失函数的标记，同时利用内容图像初始化生成图像。

- 开始非实时风格迁移的迭代训练过程。每次迭代时首先用当前的生成图像进行前向传播，其次分别计算内容损失和风格损失，然后分别进行反向传播获取内容损失和风格损失对风格迁移图像的梯度，之后利用权重系数计算相应层对风格迁移图像的梯度和，并利用 Adam 优化器使用该梯度和对风格迁移图像进行更新。每迭代若干次保存当前的风格迁移图像作为输出结果。

3.3.5.6 实验运行

根据第3.3.5.1节~第3.3.5.5节的描述补全 layer_1.py、layer_2.py、layer_3.py、style_transfer.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_chap_2_3目录下。

```

1 # 登录云平台
2 ssh root@xxx.xxx.xxx.xxx -pxxxxx
3 # 进入 code_chap_2_3_student 目录
4 cd /opt/code_chap_2_3/code_chap_2_3_student
5 # 初始化环境
6 source env.sh
7

```

2. 代码实现

补全 stu_upload 中的 layer_1.py、layer_2.py、layer_3.py、style_transfer.py 文件。

```

1 # 进入实验目录
2 cd exp_3_1_vgg
3 # 补全 layer_1.py, layer_2.py, layer_3.py, style_transfer.py
4 vim stu_upload/layer_1.py

```

```

1 # file: exp_3_3_style_transfer.py
2 if __name__ == '__main__':
3     CONTENT_LOSS_LAYERS = [ 'relu4_2' ]
4     STYLE_LOSS_LAYERS = [ 'relu1_1', 'relu2_1', 'relu3_1', 'relu4_1', 'relu5_1' ]
5     NOISE = 0.5
6     ALPHA, BETA = 1, 500
7     TRAINTEP = 2001
8     LEARNING_RATE = 1.0
9     IMAGE_HEIGHT, IMAGE_WIDTH = 192, 320
10
11    vgg = VGG19()
12    vgg.build_model()
13    vgg.init_model()
14    vgg.load_model()
15    content_loss_layer = ContentLossLayer()
16    style_loss_layer = StyleLossLayer()
17    adam_optimizer = AdamOptimizer(LEARNING_RATE, transfer_image.shape)
18
19    content_image, content_shape = vgg.load_image('content.jpg', IMAGE_HEIGHT,
20                                                 IMAGE_WIDTH)
21    style_image, _ = vgg.load_image('style.jpg', IMAGE_HEIGHT, IMAGE_WIDTH)
22    content_layers = vgg.forward(content_image, CONTENT_LOSS_LAYERS)
23    style_layers = vgg.forward(style_image, STYLE_LOSS_LAYERS)
24    transfer_image = get_random_img(content_image, NOISE)
25
26    for step in range(TRAINTEP):
27        transfer_layers = vgg.forward(transfer_image, CONTENT_LOSS_LAYERS +
28                                      STYLE_LOSS_LAYERS)
29        content_loss = np.array([])
30        style_loss = np.array([])
31        content_diff = np.zeros(transfer_image.shape)
32        style_diff = np.zeros(transfer_image.shape)
33        for layer in CONTENT_LOSS_LAYERS:
34            # TODO: 计算内容损失的前向传播
35            current_loss = _____
36            content_loss = np.append(content_loss, current_loss)
37            # TODO: 计算内容损失的反向传播
38            dloss = content_loss_layer.backward(transfer_layers[layer], content_layers[
39                layer])
40            content_diff += _____
41        for layer in STYLE_LOSS_LAYERS:
42            # TODO: 计算风格损失的前向传播
43            current_loss = _____
44            style_loss = np.append(style_loss, current_loss)
45            # TODO: 计算风格损失的反向传播
46            dloss = style_loss_layer.backward(transfer_layers[layer], style_layers[layer])
47            style_diff += _____
48        total_loss = ALPHA * np.mean(content_loss) + BETA * np.mean(style_loss)
49        image_diff = ALPHA * content_diff / len(CONTENT_LOSS_LAYERS) + BETA * style_diff /
50                     len(STYLE_LOSS_LAYERS)
51        # TODO: 利用 Adam 优化器对风格迁移图像进行更新
52        transfer_image = _____
53        if step % 20 == 0:
54            print('Step %d, loss = %f' % (step, total_loss), content_loss, style_loss)
55            vgg.save_image(transfer_image, content_shape, 'output/output_' + str(step) +
56                           '.jpg')

```

图 3.21 非实时风格迁移的完整流程实现示例

```

5 vim stu_upload/layer_2.py
6 vim stu_upload/layer_3.py
7 vim stu_upload/style_transfer.py
8

```

3. 运行实验

```

1 # 运行完整实验
2 python main_exp_3_3.py
3

```

3.3.6 实验评估

为验证实验的正确性,选择图??中所示的梵高的名画“星月夜”作为风格图像,选择如图3.23所示的风景图片作为内容图像,进行非实时风格迁移。初始化后的生成图像如图3.24所示,由于是在内容图像上加入高斯噪声所得,该初始化图像视觉上与目标内容图像很相似,同时又有一些模糊。训练迭代20次后的风格迁移图像如图3.25所示,相对于初始化图像,此时的生成图像上出现了一些类似风格图像中的油画的颜色和纹理。训练迭代1000次后的风格迁移图像如图3.26所示,该图像保留了内容图像中的大部分内容信息,如河流、河流中的船、河流两旁的楼房等,同时又具有风格图像的风格,主要是颜色和纹理,如大面积的深蓝色和天空中黄色的旋转的云朵,呈现出较好的风格迁移效果。



图 3.22 目标风格图像示例



图 3.23 目标内容图像示例

本实验的评估标准设定如下:

- 60分标准: 正确实现利用四重循环计算卷积层和池化层的前向传播和反向传播过程。给定卷积层和最大池化层的前向传播和反向传播输入矩阵和参数值,可以得到正确的前向传播输出矩阵和反向传播输出梯度。同时分别给出卷积层和最大池化层正确的前向传播和反向传播时间。
- 80分标准: 正确实现内容损失函数和风格损失函数的计算,计算得出风格迁移后的图片。给定生成图像、目标内容图像和目标风格图像,可以计算得到正确的内容损失值和



图 3.24 初始风格迁移图像示例



图 3.25 迭代 20 次后的风格迁移图像示例

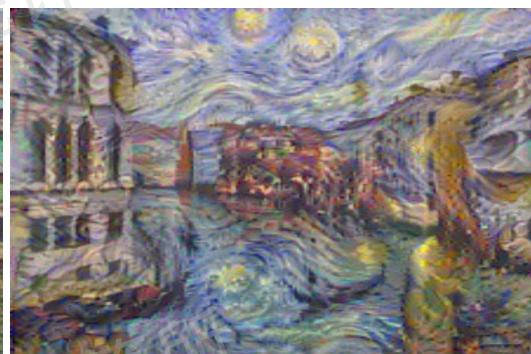


图 3.26 迭代 1000 次后的风格迁移图像示例

风格损失值；可以得到正确的内容损失和风格损失对生成图像的更新梯度，生成风格迁移后的图像。

- 100 分标准：对第 3.3 节实验中介绍的卷积层和池化层的实现中使用的四重循环进行改进，提升计算速度。给定卷积层和池化层的前向传播和反向传播输入矩阵和参数值，可以得到正确的前向传播输出矩阵和反向传播输出梯度。同时分别给出卷积层和池化层正确的前向传播和反向传播时间和对应的加速比。

3.3.7 实验思考

- 1) 使用四重循环计算卷积层前向传播和反向传播的速度较慢，如何利用高效的矩阵处理库，将四重循环中卷积核与特征图的内积运算（即向量运算）转化为矩阵运算，从而减少循环次数，加速卷积层的运算速度？

- 2) 统计训练过程中每次迭代时每层的前向传播和反向传播时间及其在每层的时间占比，哪些层的时间占比较多？前向传播和反向传播的计算瓶颈在哪些层？

- 3) 在第3.1和3.2节的实验评估中，均使用给定输入值并与正确的输出值进行比较的方式来确定某个层的实现是否正确。如果正确的输出值无法获知，如何从梯度的定义角度检查层的实现是否正确？（可参考由梯度定义引申出的梯度的数值近似实现方法）

- 4) 风格迁移的结果通常是由人直接进行判断，这是一种主观的定性判断方式，不同的人判断结果可能会有较大偏差。如何设计合理的定量判断方法，可以较为客观的评价风格迁移结果的优劣？

3.3.8 延伸拓展

通过实践本实验并分析每一层前向传播和反向传播消耗的时间，会发现风格迁移实验的计算瓶颈主要在卷积层。这主要是由于卷积层承担了卷积神经网络主要的计算量。同时，图3.3和图3.14中的卷积层前向传播和反向传播示例中均使用了四重循环进行计算，速度较慢。由于循环是一种序列化的计算过程，在计算卷积某一个位置的结果时，必须等待上一个位置的计算结束。这种序列化的计算过程浪费了大量的计算资源，因此速度很慢。那么如何能够对卷积的计算进行加速？以前向传播为例，通过分析公式(3.3)可知，卷积前向传播时，输出特征图的每个位置结果都是使用输入特征图与卷积核做矩阵内积并与偏置相加得到。因此输出特征图不同位置的计算过程本身没有相互依赖关系，可以独立计算，并且不同位置的运算过程是完全相同的，只是使用的输入数据不同（不同输出位置使用不同输入位置的数据和相同的卷积核进行计算）。这意味着卷积运算是可以高度并行化的，不同输出位置的运算可以并行进行。相比序列化的串行操作，并行化计算可以极大的提高卷积层的计算速度，减少卷积计算消耗的时间。

考虑到本实验使用 Python 实现，可以方便的使用 Numpy 实现并行化，将需要并行化的计算过程全部转变为向量化运算，在计算时 Numpy 会自动调用多个线程将向量化后的操作并行执行。以卷积的前向传播为例，分析公式(3.3)可知输出特征图的每个位置结果都是通过内积和相加操作得到。因此，可以将待计算的输入特征图不同位置的数据重排列为矩阵形式，并行化不同位置的内积操作可转变为矩阵相乘操作。对矩阵计算加偏置的操作时，Numpy 会自动对偏置向量进行广播，加到内积结果矩阵的每一列中。最后将计算获得的结果重排列为输出特征图的维度。通过利用 Numpy 的并行化优势，可以实现与四重循环完全等价的运算，但极大的提升了卷积层前向传播的计算速度。程序示例如图3.27所示。

与卷积层的前向传播并行化过程类似，也可以将卷积层的反向传播和池化层的前向、反向传播过程进行向量化，然后利用 Numpy 的并行化特性实现加速。

图3.27主要利用了 Numpy 调用 CPU 的多线程实现卷积运算的并行化，如果使用 GPU 运算卷积，可以将不同输出位置的运算分配在 GPU 不同的运算核上进行，同样可以实现卷积层的并行运算。由于 GPU 的运算核数量远多于 CPU 的线程数，GPU 并行运算卷积的速度也比 CPU 快很多，为加快运算速度，目前很多卷积神经网络的训练和推理都是在 GPU 上进行的。上述的卷积优化方法简单的来说其实就是通过 img2col 方法将卷积核转化为行向量，将对应的局部数据变为列向量，从而把卷积运算转化为了矩阵乘运算，这样就可以大大提高卷积运算的并行度以提升性能。而 DLP 的架构与 GPU 和 CPU 不同，DLP 内部包含了大量的可并行运行的 DLP-S 核^[1]，每个 DLP-S 核内部的运算单元可以直接完成矩阵的卷积计算，不需要额外的 img2col 操作来优化卷积；此外，DLP 为具有不同访存特征的数据流设计的专用通路，进一步提升了访存效率，因此相比 CPU 和 GPU 会拥有更好的能效表现。

```
1 # file: layer_2.py
2 def forward_speedup(self, input): # 前向传播的并行化计算
3     self.input = input # [N, C, H, W]
4     height = self.input.shape[2] + self.padding * 2
5     width = self.input.shape[3] + self.padding * 2
6     self.input_pad = np.zeros([self.input.shape[0], self.input.shape[1], height, width])
7     self.input_pad[:, :, self.padding:self.padding+self.input.shape[2], self.padding:
8         self.padding+self.input.shape[3]] = self.input
9     self.height_out = (height - self.kernel_size) / self.stride + 1
10    self.width_out = (width - self.kernel_size) / self.stride + 1
11    self.weight_reshape = np.reshape(self.weight, [-1, self.channel_out]) # 对卷积核进行
12        向量化
13    self.img2col = np.zeros([self.input.shape[0]*self.height_out*self.width_out, self.
14        channel_in*self.kernel_size*self.kernel_size])
15    # 对卷积层的输入特征图进行向量化重排列
16    for idxn in range(self.input.shape[0]):
17        for idxh in range(self.height_out):
18            for idxw in range(self.width_out):
19                self.img2col[idxn*self.height_out*self.width_out + idxh*self.width_out +
20                    idxw, :] = self.input_pad[idxn, :, idxh*self.stride:idxh*self.stride+self.
21                        kernel_size, idxw*self.stride:idxw*self.stride+self.kernel_size].reshape([-1])
22    # 计算卷积层的前向传播, 特征图与卷积核的内积转变为矩阵相乘, 再加偏置
23    output = np.dot(self.img2col, self.weight_reshape) + self.bias
24    self.output = output.reshape([self.input.shape[0], self.height_out, self.width_out,
25        -1]).transpose([0, 3, 1, 2]) # 对卷积层的输出结果进行重排列
26    return self.output
```

图 3.27 卷积层的并行化实现示例

第4章 编程框架实践

在第2~3章，我们使用高级编程语言 Python 实现了卷积、池化、ReLU 等深度学习算法中的常用操作，并最终实现了非实时风格迁移算法。在深度学习算法中，诸如卷积、池化、全连接等基本操作会被大量、重复地使用，而编程框架将这些基本操作封装成了一系列组件，从而帮助程序员更简单地实现已有算法或设计新的算法。

目前常用的深度学习编程框架有十多种，而 TensorFlow 是其中最主流、应用最广泛的编程框架之一。TensorFlow 向上提供了一系列高性能的 API，能够高效的实现各类深度学习算法；向下能够运行在包括 CPU、GPU 和 DLP 等在内的多种硬件平台上，具有良好的跨平台特性。

本章首先以 VGG19 为例，介绍如何使用 TensorFlow 在 CPU 及 DLP 平台上实现图像分类；之后介绍如何使用 TensorFlow 在 CPU 及 DLP 平台上实现实时风格迁移算法的推断；随后介绍实时风格迁移算法训练的实现过程；最后介绍如何在 TensorFlow 中新增用户自定义算子，并将其集成到已经训练好的风格迁移网络中。

4.1 基于 VGG19 实现图像分类

4.1.1 实验目的

掌握 TensorFlow 编程框架的使用，能够在 CPU 平台上使用 TensorFlow 编程框架实现基于 VGG19 网络的图像分类，并在深度学习处理器 DLP 上完成图像分类。具体包括：

- 1) 掌握使用 TensorFlow 编程框架处理深度学习任务的流程；
- 2) 熟悉 TensorFlow 中常用数据结构的使用方法；
- 3) 掌握 TensorFlow 中常用 API 的使用方法，包括卷积、激活等相关操作；
- 4) 与第3章的实验比较，理解使用编程框架实现深度学习算法的便捷性及高效性。

实验工作量：约 30 行代码，约需 2 个小时。

4.1.2 背景介绍

4.1.2.1 TensorFlow

TensorFlow 是由谷歌团队开发并于 2015 年 11 月开源的深度学习框架^{[9][10]}，用于实施和部署大规模机器学习模型。其在功能、性能、灵活性等方面具有诸多优势，能够支持深度学习算法在 CPU、GPU 和 DLP 等硬件平台上的部署，并支持大规模的神经网络模型。

TensorFlow 提供了一系列高性能的 API，方便程序员高效地实现深度学习算法。以目前较为常用的卷积神经网络 VGG16^[1] 为例，对每个卷积层，首先输入与权重做卷积运算^[11]，然后加上偏置，最后通过非线性激活函数 ReLU 输出。在第2章中使用高级编程语言实现了上述步骤，而在 TensorFlow 中则提供了一系列封装好的 API，方便地实现上述操作。实现

VGG19 所需的主要函数的使用方法及参数含义如表 4.1 所示^①。

TensorFlow 使用 Python 作为开发语言，并支持如 NumPy、SciPy 等多个 Python 扩展程序库以高效处理多种类型数据的计算等工作。例如：当需要读取以.mat 文件格式保存的网络参数时，通常会使用 SciPy 库中的 scipy.io 模块^[12]；而当需要做图像相关处理时，通常会使用 SciPy 库中的 scipy.misc 模块^[13]来处理图像 io 相关的操作。这两个模块中常用函数的使用方法及参数含义如表 4.2 所示。

TensorFlow 使用计算图来表示深度学习算法的网络拓扑结构。在进行深度学习训练时，每次均会有一个训练样本作为计算图的输入。如果每次的训练样本都用常量表示的话，就需要把所有训练样本都作为常量添加到 TensorFlow 的计算图中，这会导致最后的计算图急剧膨胀。

为了解决计算图膨胀的问题，TensorFlow 中提供了占位符机制。占位符是 TensorFlow 中特有的数据结构，它本身没有初值，仅在程序中分配了内存。占位符可以用来表示模型的训练样本，在创建时会在计算图中增加一个节点，且只需在执行会话时填充占位符的值即可。TensorFlow 中使用 tf.placeholder() 来创建占位符，并需要指明其数据类型 dtype，即填充数据的数据类型。占位符的输入参数还有 shape，即填充数据的形状；name，即该占位符在计算图中的名字。其中，dtype 为必填参数，而 shape 和 name 则均为可选参数。使用时需要在会话中与 feed_dict 参数配合，用 feed_dict 参数来传递待填充的数据给占位符。

4.1.2.2 量化工具

深度学习模型需要量化并存储为.pb 格式的文件，才可以在 TensorFlow 框架下运行在 DLP 平台上。在第2.2.2.1已经介绍过量化的具体原理，这里重点介绍相应量化工具的相关背景。具体而言，本实验平台提供了 TensorFlow 框架下的量化工具 fppb_to_intpb，用于将 Float32 类型的模型文件量化为 INT8 或者 INT16 的模型文件。

以 VGG19 为例，该量化工具的使用方式如下：

```
1 python fppb_to_intpb.py vgg19_int8.ini
```

其中，vgg19_int8.ini 为参数配置文件，描述了量化前后的模型文件路径、量化位宽等信息。具体内容如下所示：

```
1 [preprocess]
2 mean = 123.68, 116.78, 103.94      ; 均值，顺序依次为 mean_r、mean_g、mean_b
3 std = 1.0                            ; 方差
4 color_mode = rgb                     ; 网络的输入图片是 rgb、bgr、grey
5 crop = 224, 224                      ; 前处理最终将图片处理为 224 * 224 大小
6 calibration = default_preprocess_cali ; 校准数据读取及前处理的方式，可以根据需求进行自定义，[preprocess] 和 [data] 中定义的参数均为 calibrate_data.py 的输入参数
7
8 [config]
9 activation_quantization_alg = naive   ; 输入量化模式，可选 naive 和 threshold_search，naive 为基础模式，threshold_search 为阈值搜索模式
10 device_mode = clean                  ; 可选 clean、mlu 和 origin，建议使用 clean，使用 clean 生成的模型在运行时会自动选择可运行的设备
```

^①该部分参考自 TensorFlow 官方 github: https://github.com/tensorflow/docs/tree/r1.14/site/en/api_docs/python/tf/nn

表 4.1 TensorFlow 中卷积计算的常用函数

函数名	功能描述	参数介绍
<code>tf.nn.conv2d(input, filter=None, strides=None, padding=None, use_cudnn_on_gpu=True, data_format='NHWC', dilations=[1, 1, 1, 1], name=None, filters=None)</code>	计算输入张量 <code>input</code> 和卷积核 <code>filter</code> 的卷积，返回卷积计算的结果张量。	<code>input</code> : 输入张量，仅支持 <code>half</code> 、 <code>bfloat16</code> 、 <code>float32</code> 、 <code>float64</code> 类型。 <code>filter</code> : 卷积核，数据类型需与 <code>input</code> 一致。 <code>strides</code> : 卷积步长。 <code>padding</code> : 边界扩充，其值为“ <code>SAME</code> ”或“ <code>VALID</code> ”。“ <code>SAME</code> ”表示对输入先进行边界扩充再进行卷积运算；“ <code>VALID</code> ”表示不做边界扩充，直接从每行输入的第一个像素开始做卷积运算，对于每行参与卷积的最后一段输入，尺寸小于卷积核的部分直接舍弃。 <code>use_cudnn_on_gpu</code> : 布尔值，缺省为 <code>True</code> 。 <code>data_format</code> : 输入和输出数据的数据格式，其值为“ <code>NHWC</code> ”或“ <code>NCHW</code> ”。缺省为“ <code>NHWC</code> ”，表示数据存储格式为：[batch, height, width, channels]。 <code>dilations</code> : 输入张量在每个维度上的膨胀系数，其值为整数或者长度为 1、2 或 4 的整数数列。 <code>name</code> : 可选参数，表示操作的名称。 <code>filters</code> : 同 <code>filter</code> 。
<code>tf.nn.bias_add(value, bias, data_format=None, name=None)</code>	对输入张量 <code>value</code> 加上偏置 <code>bias</code> ，并返回一个与 <code>value</code> 相同类型的张量。	<code>value</code> : 输入张量。其数据类型包括 <code>float</code> 、 <code>double</code> 、 <code>int64</code> 、 <code>int32</code> 、 <code>uint8</code> 、 <code>int16</code> 、 <code>int8</code> 、 <code>complex6</code> 或 <code>complex128</code> 。 <code>bias</code> : 一阶张量，其形状 (<code>shape</code>) 与 <code>value</code> 的最后一阶一致，数据类型需与 <code>value</code> 一致（量化类型除外）。由于该函数支持广播形式，因此 <code>value</code> 可以有任意形状。 <code>data_format</code> : 输入张量的数据格式。 <code>name</code> : 可选参数，表示操作的名称。
<code>tf.nn.relu(features, name=None)</code>	对输入张量 <code>features</code> 计算 ReLU，返回一个与 <code>features</code> 相同数据类型的张量。	<code>features</code> : 输入张量，其数据类型包括 <code>float32</code> 、 <code>float64</code> 、 <code>int32</code> 、 <code>uint8</code> 、 <code>int16</code> 、 <code>int8</code> 、 <code>int64</code> 、 <code>bfloat16</code> 、 <code>uint16</code> 、 <code>half</code> 、 <code>uint32</code> 、 <code>uint64</code> 或 <code>qint8</code> 。 <code>name</code> : 可选参数，表示操作的名称。
<code>tf.nn.softmax(logits, axis=None, name=None, dim=None)</code>	对输入张量 <code>logits</code> 执行 softmax 激活操作，返回一个与 <code>logits</code> 相同数据类型、形状的张量。	<code>logits</code> : 输入张量，其数据类型包括 <code>half</code> 、 <code>float32</code> 、 <code>float64</code> 。 <code>axis</code> : 执行 softmax 操作的维度，缺省为 -1，表示最后一个维度。 <code>name</code> : 可选参数，表示操作的名称。
<code>tf.nn.max_pool(value, ksize, strides, padding, data_format='NHWC', name=None, input=None)</code>	对输入张量 <code>value</code> 执行最大池化操作，返回操作的结果。	<code>value</code> : 输入张量，其数据格式由 <code>data_format</code> 定义。 <code>ksize</code> : 对输入张量的每个维度执行最大池化操作的窗口尺寸。 <code>strides</code> : 对输入张量的每个维度执行最大池化操作的滑动步长。 <code>padding</code> : 边界扩充，其值为“ <code>SAME</code> ”或“ <code>VALID</code> ”。 <code>data_format</code> : 输入和输出数据的数据格式，支持“ <code>NHWC</code> ”、“ <code>NCHW</code> ”及“ <code>NCHW_VECT_C</code> ”格式。 <code>name</code> : 可选参数，表示操作的名称。 <code>input</code> : 同 <code>value</code> 。
<code>tf.nn.conv2d_transpose(value=None, filter=None, output_shape=None, strides=None, padding='SAME', data_format='NHWC', name=None, input=None, filters=None, dilations=None)</code>	计算输入张量 <code>value</code> 和卷积核 <code>filter</code> 的转置卷积，返回计算的结果张量。	<code>value</code> : 转置卷积计算的输入张量，数据类型为 <code>float</code> ，数据格式可以是“ <code>NHWC</code> ”或“ <code>NCHW</code> ”。 <code>filter</code> : 卷积核，数据类型需与 <code>value</code> 一致。 <code>output_shape</code> : 转置卷积的输出形状。 <code>strides</code> : 卷积步长。 <code>padding</code> : 边界扩充，其值为“ <code>SAME</code> ”或“ <code>VALID</code> ”。 <code>data_format</code> : 输入和输出数据的数据格式，其值为“ <code>NHWC</code> ”或“ <code>NCHW</code> ”。缺省为“ <code>NHWC</code> ”，表示数据存储格式为：[batch, height, width, channels]。 <code>name</code> : 可选参数，表示返回的张量名称。 <code>input</code> : 同 <code>value</code> 。 <code>filters</code> : 同 <code>filter</code> 。 <code>dilations</code> : 输入张量在每个维度上的膨胀系数，其值为整数或者长度为 1、2 或 4 的整数数列。

表 4.2 常用的 `scipy.io` 及 `scipy.misc` 函数

函数名	功能描述	参数介绍
<code>scipy.io.loadmat(file_name, mdict=None, appendmat=True, **kwargs)</code>	装载 MATLAB 文件 (.mat)，返回以变量名为键、以加载的矩阵为值的字典，格式为 (mat_dict=dict)。	<code>file_name</code> : .mat 文件的名称。 <code>mdict</code> : 可选参数，插入了.mat 文件所列变量的字典。 <code>appendmat</code> : 可选参数，布尔类型。为 True 表示将.mat 扩展名添加到 <code>file_name</code> 之后。 其余参数含义请参考 ^[14] 。
<code>scipy.misc.imread(name, flatten=False, mode=None)</code>	从文件 <code>name</code> 中读入一张图像，将其处理成 ndarray 类型的数据并返回。	<code>name</code> : 待读取的文件名称。 <code>flatten</code> : 布尔类型。其值为 True 表示将彩色层扁平化处理成单个灰度层。 <code>mode</code> : 表示将图像转换成何种模式，其值可以是 "L"、"P"、"RGB"、"RGBA"、"CMYK"、"YCbCr"、"I"、"F" 等。
<code>scipy.misc.imresize(arr, size, interp='bilinear', mode=None)</code>	对图像 <code>arr</code> 的尺寸进行缩放，返回处理后的 ndarray 类型数据。	<code>arr</code> : 待缩放的图像，数据类型为 ndarray。 <code>size</code> : 可以是 int、float 或 tuple 类型。为 int 类型时表示将图像缩放到当前尺寸的百分比；为 float 类型时表示将图像缩放到当前尺寸的几倍；为 tuple 类型时表示缩放后的图像尺寸。 <code>interp</code> : 用于缩放的插值方法，其值可以是 "nearest"、"lanczos"、"bilinear"、"bicubic" 或 "cubic" 等。 <code>mode</code> : 缩放前需将输入图像转换成何种图像模式，其值可以是 "L"、"P" 等。
<code>scipy.misc.imsave(name, arr, format=None)</code>	将 ndarray 类型的数组 <code>arr</code> 保存为图像 <code>name</code> 。	<code>name</code> : 输出的图像文件名称。 <code>arr</code> : 待保存的 ndarray 类型数组。 <code>format</code> : 保存的图像格式。

```

11 use_convfirst = False           ; 是否使用 convfirst
12 quantization_type = int8     ; 量化位宽，目前可选 int8 和 int16
13 debug = False
14 weight_quantization_alg = naive ; 权值量化模式，可选 naive 和 threshold_search，naive 为基础模式，threshold_search 为阈值搜索模式
15 int_op_list = Conv, FC, LRN   ; 要量化的 layer 的类型，目前可量化 Conv、FC 和 LRN
16 channel_quantization = False ; 是否使用分通道量化
17
18 [model]
19 output_tensor_names = Softmax:0 ; 输入 Tensor 的名字，可以是多个，以逗号隔开
20 original_models_path = ./vgg19.pb ; 输入 pb
21 save_model_path = ./vgg19_int8.pb ; 输出 pb
22 input_tensor_names = img_placeholder:0 ; 输出 Tensor 的名字，可以是多个，以逗号隔开
23
24 [data]
25 num_runs = 1                  ; 运行次数，比如 batch_size = 2, num_runs = 10 则表示使用
        data_path 指定的数据集中的前 20 张图片作为校准数据
26 data_path = ./image_list       ; 数据文件存放路径
27 batch_size = 1                ; 每次运行的 batch_size

```

描写该参数配置文件时，需注意以下几点：

1. 关于 `color_mode`:

如果 `color_mode` 是 `grey` (即灰度图模式)，则 `mean` 只需要传入一个值。

2. 关于 `activation_quantization_alg` 和 `weight_quantization_alg`:

`threshold_search` 阈值搜索模式用于处理存在异常值的待量化数据集，该模式能够过滤部分异常值，重新计算出数据集的最值，用最新值来计算数据集的量化参数，从而提高数

据集整体的量化质量。对于不存在异常值且数据分布紧凑的情况，不建议使用该算法，比如权重的量化。

3. 关于 device_mode:

mlu: 将输出 pb 格式模型文件的所有节点的 device 设置为 MLU。

clean: 将输出 pb 格式模型文件所有节点的 device 清除，运行时可根据算子注册情况自动选择可运行的设备。

origin: 使用和输入 pb 格式模型文件一样的设备指定(在配置文件 config 部分 int_op_list 参数中指定的算子除外，如上文 vgg19_int8.ini 示例中的 Conv, FC 和 LRN)。

4. 关于 use_convfirst:

use_convfirst 是针对第一层卷积的优化选项，可用于提升网络的整体性能。如果网络要使用 convfirst 优化，需满足以下几个条件：

- (a) 网络的前处理不能包含在 graph 中；
- (b) 网络的前处理必须是以下形式或者可以转换为以下形式： $\text{input} = (\text{input} - \text{mean}) / \text{std}$ ；
- (c) 网络的第一层必须是 Conv2D，输入图片必须是 3 通道；
- (d) 必须在输入 ini 文件中的 [preprocess] 下定义 mean、std 和 color_mode。

4.1.3 实验环境

本节实验所涉及的硬件平台和软件环境如下：

- 硬件平台：CPU、DLP
- 软件环境：TensorFlow 1.14, Python 编译环境及相关的扩展库，包括 Python 2.7.12, Pillow 4.2.1, Scipy 1.0.0, NumPy 1.16.6、CNML 高性能算子库、CNRT 运行时库

4.1.4 实验内容

利用 TensorFlow 的 API，实现第3.1节中基于 VGG19 进行图像分类的实验，运行的平台包括 CPU 和 DLP。最后比较两种平台实现的差异。

4.1.5 实验步骤

本实验主要包含以下步骤：读取图像、定义操作、定义网络结构、CPU 上实现、DLP 上实现、实验运行与对比。

4.1.5.1 读取图像

首先读入待计算的图像。在本实验中，利用 `scipy.misc` 模块内置的函数读入待处理图像，并将图像处理成便于数值计算的 `ndarray` 类型。程序示例如图4.1所示。

4.1.5.2 定义卷积层、池化层

分别定义卷积层、池化层的操作步骤，如图 4.2所示。

```
1 # file: evaluate_cpu.py
2 import scipy.misc
3 import numpy as np
4 import time
5 import tensorflow as tf
6
7 os.putenv('MLU_VISIBLE_DEVICES','')#设置MLU_VISIBLE_DEVICES=""来屏蔽DLP
8
9 def load_image(path):
10 # TODO: 使用scipy.misc模块读入输入图像，调用preprocess函数对图像进行预处理，并返回形状为(1,244,244,3)的数组image
11     mean = np.array([123.68, 116.779, 103.939])
12     image = _____
13     _____
14     return image
15
16 def preprocess(image,mean):
17     return image - mean
18
```

图 4.1 读取图像作为输入

```
1 # file: evaluate_cpu.py
2 def _conv_layer(input, weights, bias):
3 # TODO: 定义卷积层的操作步骤，input为输入张量，weights为权重，bias为偏置，返回计算的结果
4     _____
5
6 def _pool_layer(input):
7 # TODO: 定义最大池化的操作步骤，input为输入张量，返回最大池化操作后的计算结果
8     _____
9
```

图 4.2 定义卷积层、池化层

4.1.5.3 定义 VGG19 网络结构

为方便对比，采用与第3.3节相同的预训练模型及层命名，逐层定义需要执行的操作，每一层输出作为下一层输入，从而搭建起完整的 VGG19 网络。程序示例如图 4.3 所示。如图 4.3 所示。

```

1 # file: evaluate_cpu.py
2 def net(data_path, input_image):
3     # 该函数定义VGG19网络结构, data_path为预训练好的模型文件,
4         # input_image为待分类的输入图像, 该函数定义43层的VGG19网络结构
5         # net并返回该网络
6     layers = (
7         'conv1_1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1',
8
9         'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2', 'pool2',
10        'conv3_1', 'relu3_1', 'conv3_2', 'relu3_2', 'conv3_3',
11        'relu3_3', 'conv3_4', 'relu3_4', 'pool3',
12        'conv4_1', 'relu4_1', 'conv4_2', 'relu4_2', 'conv4_3',
13        'relu4_3', 'conv4_4', 'relu4_4', 'pool4',
14        'conv5_1', 'relu5_1', 'conv5_2', 'relu5_2', 'conv5_3',
15        'relu5_3', 'conv5_4', 'relu5_4', 'pool5',
16        'fc6', 'relu6', 'fc7', 'relu7', 'fc8', 'softmax' )
17
18
19     data = scipy.io.loadmat(data_path)
20     weights = data['layers'][0]
21
22     net = {}
23     current = input_image
24     for i, name in enumerate(layers):
25         if name[:4] == 'conv':
26             # TODO: 从模型中读取权重、偏置, 执行卷积计算, 结果存入 current
27
28             -----
29             elif name[:4] == 'relu':
30                 # TODO: 执行 ReLU 计算, 结果存入 current
31
32                 -----
33                 # TODO: 完成其余层的定义, 最终结果存入 current
34
35             -----
36             net[name] = current
37
38             assert len(net) == len(layers)
39             return net
40
41
42 def preprocess(image, mean):
43     return image - mean

```

图 4.3 定义 VGG19 网络结构

4.1.5.4 CPU 平台上利用 VGG19 网络实现图像分类

在 TensorFlow 的会话中，利用前面定义好的 VGG19 网络，实现对输入图像的分类。程序示例如图 4.4 所示：

```

1 # file: evaluate_cpu.py
2 IMAGE_PATH = 'cat1.jpg' VGG_PATH = 'imagenet-vgg-verydeep-19.mat'
3
4 if __name__ == '__main__':
5     input_image = load_image(IMAGE_PATH)
6
7     with tf.Session() as sess:
8         img_placeholder = tf.placeholder(tf.float32, shape
9             =(1,224,224,3),
10            name='img_placeholder')
11        # TODO: 调用 net 函数, 生成 VGG19 网络模型并保存在 nets 中
12        nets = _____
13        for i in range(10):
14            start = time.time()
15            _____
16            end = time.time()
17            delta_time = end - start
18            print("processing time: %s" % delta_time)
19
20        prob = preds['softmax'][0]
21        top1 = np.argmax(prob)
22        print('Classification result: id = %d, prob = %f' % (top1,
23            prob[top1]))

```

图 4.4 利用 VGG19 网络实现图像分类

4.1.5.5 DLP 平台上利用 VGG19 网络实现图像分类

DLP 的机器学习编程序 CNML 已集成到 TensorFlow 框架中，与 CPU 上的实验类似，可以直接利用前面定义好的 VGG19 网络来实现图像分类。由于 DLP 平台上仅支持量化过的深度学习模型，首先需要将原始模型文件保存为 pb 格式，然后调用集成到 TensorFlow 中的量化工具将模型参数量化为 INT8 数据类型并形成新的 pb 格式模型文件。此外，需要在程序中设置 DLP 的核数、数据精度等运行参数，以最大发挥 DLP 的性能，这部分可以通过 config.mlu_options 进行配置。最后，编译运行 Python 程序得到图像分类结果。

1. 将模型文件保存为 pb 格式

在会话部分添加部分代码，保存模型为 vgg19.pb 文件。程序示例如 4.5 所示。

2. 模型量化

生成的 vgg19.pb 模型需要经过量化后才可以在 DLP 上运行，所以先将原始的 Float32 数据类型的 pb 模型量化成为 INT 类型。在 vgg19/fppb_to_intpb 目录下运行以下命令，使用量化工具完成对模型的量化，生成新模型 vgg19_int8.pb。

```
1 python fppb_to_intpb.py vgg19_int8.ini
```

3. 设置 DLP 运行环境

在文件 evaluate_mlu.py 中设置程序在 DLP 上运行需要的环境参数如核数和数据精度等。程序示例如下：

```
1 # file: evaluate_dlp.py
2 import numpy as np
```

```
1 # file: evaluate_dlp.py
2 import numpy as np
3 import struct
4 import os
5 import scipy.io
6 import time
7 import tensorflow as tf
8 from tensorflow.python.framework import graph_util #用于将模型文件保存为 pb 格式
9
10 if __name__ == '__main__':
11     input_image = load_image(IMAGE_PATH)
12
13     with tf.Session() as sess:
14         # TODO: 代码见上一小节
15
16         -----
17
18         prob = preds['softmax'][0]
19         top1 = np.argmax(prob)
20         print('Classification result: id = %d, prob = %f' % (top1, prob[top1]))
21
22         print('*** Start Saving Frozen Graph ***')
23         # We retrieve the protobuf graph definition
24         input_graph_def = sess.graph.as_graph_def()
25         output_node_names = ["Softmax"]
26         # We use a built-in TF helper to export variables to constant
27         output_graph_def = graph_util.convert_variables_to_constants(
28             sess,
29             input_graph_def,
30             output_node_names,
31         )
32         # Finally we serialize and dump the output graph to the filesystem
33         with tf.gfile.GFile("vgg19.pb", "wb") as f:
34             f.write(output_graph_def.SerializeToString())
35         print("**** Save Frozen Graph Done ****")
```

图 4.5 将模型文件保存为 pb 格式

```
3 import struct
4 import os
5 import scipy.io
6 import time
7 import tensorflow as tf
8 from tensorflow.python.framework import graph_util
9
10 os.putenv('MLU_VISIBLE_DEVICES','0')#设置程序运行在DLP上
11
12 IMAGE_PATH = 'cat1.jpg'
13 VGG_PATH = 'vgg19_int8.pb'
14
15 if __name__ == '__main__':
16     input_image = load_image(IMAGE_PATH)
17
18     g = tf.Graph()
19
20     # setting mlu configurations
21     config = tf.ConfigProto(allow_soft_placement=True,
22                            inter_op_parallelism_threads=1,
23                            intra_op_parallelism_threads=1)
24     config.mlu_options.data_parallelism = 1
25     config.mlu_options.model_parallelism = 1
26     config.mlu_options.core_num = 16 # 1 4 16
27     config.mlu_options.core_version = "MLU270"
28     config.mlu_options.precision = "int8"
29     config.mlu_options.save_offline_model = False
30
31     model = VGG_PATH
32
33     with g.as_default():
34         with tf.gfile.FastGFile(model, 'rb') as f:
35             graph_def = tf.GraphDef()
36             graph_def.ParseFromString(f.read())
37             tf.import_graph_def(graph_def, name='')
38
39         with tf.Session(config=config) as sess:
40             sess.run(tf.global_variables_initializer())
41             input_tensor = sess.graph.get_tensor_by_name('img_placeholder:0')
42             output_tensor = sess.graph.get_tensor_by_name('Softmax:0')
43
44             for i in range(10):
45                 start = time.time()
46                 # TODO: 计算 output_tensor
47
48                 end = time.time()
49                 delta_time = end - start
50                 print("Inference processing time: %s" % delta_time)
51
52             prob = preds[0]
53             top1 = np.argmax(prob)
54
55             print('Classification result: id = %d, prob = %f' % (top1, prob[top1]))
```

4. 在 DLP 平台上完成图像分类

在 DLP 平台上运行以下命令，使用 VGG19 网络完成图像分类。

```
1 python main_exp_4_1.py
```

4.1.5.6 实验运行

根据第4.1.5.1节~第4.1.5.5节的描述补全 evaluate_cpu.py、evaluate_mlu.py 代码，并通过 Python 运行上述代码。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_4_student 目录下。

```

1 # 登录云平台
2 ssh root@xxx.xxx.xxx.xxx -p xxxxx
3 # 进入 /opt/code_chap_4_student 目录
4 cd /opt/code_chap_4_student
5 # 初始化环境
6 cd env
7 source env.sh

```

2. 代码实现

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

```

1 # 进入实验目录
2 cd exp_4_1_vgg19_student
3 # 补全 cpu 实现代码
4 vim stu_upload/evaluate_cpu.py
5 # 补全 mlu 实现代码
6 vim stu_upload/evaluate_mlu.py

```

3. CPU 运行

```

1 # cpu 上运行，生成pb模型，模型保存在models目录中
2 ./run_cpu.sh

```

4. DLP 运行

```

1 # 对保存的 pb 模型进行量化
2 cd fppb_to_intpb
3 python fppb_to_intpb.py vgg19_int8.ini
4 # mlu 上运行
5 ./run_mlu.sh
6 # 运行完整实验
7 python main_exp_4_1.py

```

4.1.6 实验评估

本实验的评估标准设定如下：

- 60 分标准：在 CPU 平台上正确实现读入输入图像、定义卷积层、池化层的过程。可以通过在会话中打印输入图像、卷积层计算结果和池化层计算结果来验证。

- 80 分标准：在 CPU 平台上完成网络模型的正确转换，以及网络参数的正确读取。
- 90 分标准：在 CPU 平台上正确实现对 VGG19 网络的定义，给定 VGG19 的网络参数值和输入图像，可以得到正确的 Softmax 层输出结果和正确的图像分类结果。
- 100 分标准：在云平台上正确实现对 VGG19 网络的 pb 格式转换及量化，给定 VGG19 的网络参数值和输入图像，可以得到正确的 Softmax 层输出结果和正确的图像分类结果，处理时间相比 CPU 平台平均提升 10 倍以上。

4.1.7 实验思考

- 1) 本实验与第3.3小节中使用 Python 实现的图像分类相比，在识别精度、识别速度等方面有哪些差异？为什么会有这些差异？

4.2 实时风格迁移

4.2.1 实验目的

掌握如何使用 TensorFlow 实现实时风格迁移算法中的图像转换网络的推断模块，并进行图像的风格迁移处理。具体包括：

- 1) 掌握使用 TensorFlow 定义完整网络结构的方法；
- 2) 掌握使用 TensorFlow 恢复模型参数的方法；
- 3) 以实时风格迁移算法为例，掌握在 CPU 平台上使用 TensorFlow 进行神经网络推断的方法；
- 4) 理解 DLP 高性能算子库集成到 TensorFlow 框架的基本原理；
- 5) 掌握在 DLP 平台上使用 TensorFlow 对模型进行量化并实现神经网络推断的方法。

实验工作量：约 20 行代码，约需 2 个小时。

4.2.2 背景介绍

在《智能计算系统》教材的第四章中，使用 TensorFlow 实现了一个非实时的风格迁移算法。在该算法中，对于每个输入图像，都需要通过对风格迁移图像的多次迭代训练得到风格迁移后的输出，耗时较长，实时性差。因此，Johnson 等^[15] 提出了一种实时的图像风格迁移算法。该实时风格迁移算法中包含了图像转换网络和特征提取网络，这两个网络中的所有模型参数都可以提前训练好，随后输入图像可以通过其中的图像转换网络直接输出风格迁移后的图像，基本达到实时的效果。

下面重点介绍该算法的核心图像转换网络的相关背景知识。

- 图像转换网络

图像转换网络的结构如图 4.6 所示。该网络由三个卷积层、五个残差块、两个转置卷积层再接一个卷积层构成。除了输出层，所有非残差卷积层后面都加了批归一化（batch normalization, BN）^[16] 和 ReLU 操作，输出层使用 tanh 函数将输出像素值限定在 [0, 255] 范围内；第一层和最后一层卷积使用 9×9 卷积核，其它卷积层都使用 3×3 卷积核；每个残差块中包含两层卷积。每一层的具体参数如表 4.3 所示。

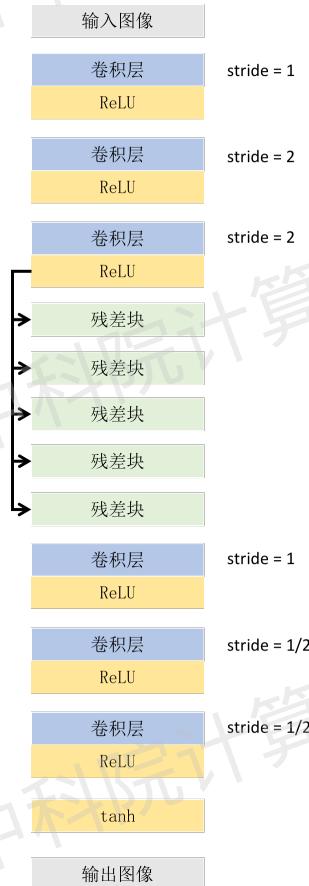


图 4.6 图像转换网络的网络结构

表 4.3 图像转换网络中使用的网络结构参数^[17]

层	规格
输入	$3 \times 256 \times 256$
反射填充 (40×40)	$3 \times 336 \times 336$
$32 \times 9 \times 9$ 卷积, 步长 1	$32 \times 336 \times 336$
$64 \times 3 \times 3$ 卷积, 步长 2	$64 \times 168 \times 168$
$128 \times 3 \times 3$ 卷积, 步长 2	$128 \times 84 \times 84$
残差块, 128 个卷积	$128 \times 80 \times 80$
残差块, 128 个卷积	$128 \times 76 \times 76$
残差块, 128 个卷积	$128 \times 72 \times 72$
残差块, 128 个卷积	$128 \times 68 \times 68$
残差块, 128 个卷积	$128 \times 64 \times 64$
$64 \times 3 \times 3$ 卷积, 步长 1/2	$64 \times 128 \times 128$
$32 \times 3 \times 3$ 卷积, 步长 1/2	$32 \times 256 \times 256$
$3 \times 9 \times 9$ 卷积, 步长 1	$3 \times 256 \times 256$

• 残差块

图像转换网络中包含了五个残差块，其基本结构如图 4.7 所示：输入 x 经过一个卷积层，再做 ReLU，然后经过另一个卷积层得到 $F(x)$ ，再加上 x 得到输出 $H(x) = F(x) + x$ ，然后做 ReLU 得到基本块的最终输出 y 。当输入 x 的维度与卷积输出 $F(x)$ 的维度不同时，需要先对 x 做恒等变换使二者维度一致，然后再加和。

与常规的卷积神经网络相比，残差块增加了从输入到输出的直连（shortcut connection），其卷积拟合的是输出与输入的差（即残差）。由于输入和输出都做了批归一化，符合正态分布，因此输入和输出可以做减法，如图 4.7 中 $F(x) = H(x) - x$ 。残差网络的优点是对数据波动更灵敏，更容易求得最优解，因此能够改善深层网络的训练。

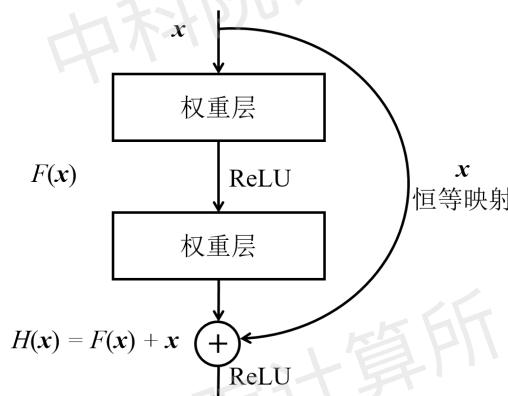


图 4.7 残差块结构

• 转置卷积

转置卷积^[19]又可以称为小数步长卷积，图 4.8 是一个转置卷积的示例。输入矩阵 InputData 是 2×2 的矩阵，卷积核 Kernel 的大小为 3×3 ，卷积步长为 1，输出 OutputData 是 4×4 的矩阵。

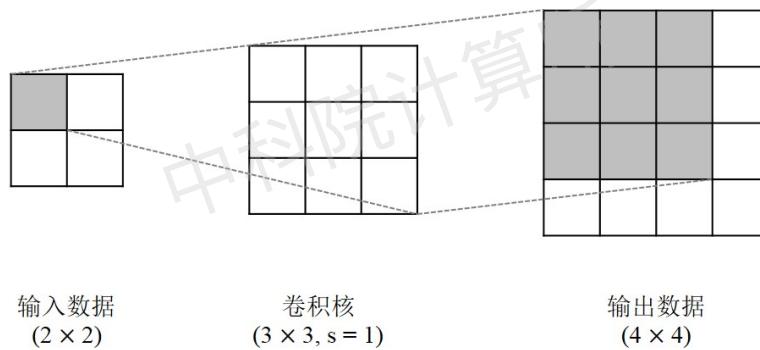


图 4.8 转置卷积

可以采用矩阵乘法来实现转置卷积，具体步骤如下：

1. 将输入矩阵 InputData 展开成为 4×1 的列向量 x 。

2. 把 3×3 的卷积核 Kernel 转换成一个 4×16 的稀疏卷积矩阵 \mathbf{W} :

$$\mathbf{W} = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \end{bmatrix}$$

其中 $w_{i,j}$ 表示卷积核 Kernel 的第 i 行第 j 列元素。

3. 求 \mathbf{W} 的矩阵转置 \mathbf{W}^\top :

$$\mathbf{W}^\top = \begin{bmatrix} w_{0,0} & 0 & 0 & 0 \\ w_{0,1} & w_{0,0} & 0 & 0 \\ w_{0,2} & w_{0,1} & 0 & 0 \\ 0 & w_{0,2} & 0 & 0 \\ w_{1,0} & 0 & w_{0,0} & 0 \\ w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\ w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\ 0 & w_{1,2} & 0 & w_{0,2} \\ w_{2,0} & 0 & w_{1,0} & 0 \\ w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\ w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\ 0 & w_{2,2} & 0 & w_{1,2} \\ 0 & 0 & w_{2,0} & 0 \\ 0 & 0 & w_{2,1} & w_{2,0} \\ 0 & 0 & w_{2,2} & w_{2,1} \\ 0 & 0 & 0 & w_{2,2} \end{bmatrix}$$

4. 转置卷积操作等同于矩阵 \mathbf{W}^\top 与向量 \mathbf{x} 的乘积: $\mathbf{y} = \mathbf{W}^\top \times \mathbf{x}$

5. 上一步骤得到的 \mathbf{y} 为 16×1 的向量, 将其形状修改为 4×4 的矩阵得到最终的结果 OutputData。

• 实例归一化

图像转换网络中, 每个卷积计算之后激活函数之前都插入了一种特殊的跨样本的批归一化层。该方法由谷歌的科学家在 2015 年提出, 它使用多个样本做归一化, 将输入归一化到加了参数的标准正态分布上。这样可以有效避免梯度爆炸或消失, 从而训练出较深的神经网络。批归一化的计算方法见公式 (4.1)。

$$y_{tijk} = \frac{x_{tijk} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}, \quad \mu_i = \frac{1}{HWN} \sum_{t=1}^N \sum_{l=1}^W \sum_{m=1}^H x_{tilm}, \quad \sigma_i^2 = \frac{1}{HWN} \sum_{t=1}^N \sum_{l=1}^W \sum_{m=1}^H (x_{tilm} - \mu_i)^2 \quad (4.1)$$

其中, x_{tijk} 表示输入图像集合中的第 $tijk$ 个元素, k, j 分别表示其在 H、W 方向的序号, t 表示输入图像在集合中的序号, i 表示特征通道序号。

批归一化方法是在输入图像集合上分别对 NHW 做归一化以保证数据分布的一致性, 而在风格迁移算法中, 由于迁移后的结果主要依赖于某个图像实例, 所以对整个输入集合

做归一化的方法并不适合。2017 年有学者针对实时风格迁移算法提出了实例归一化方法^[20]。不同于批归一化，该方法使用公式(4.2)来对 HW 做归一化，从而保持每个图像实例之间的独立，在风格迁移算法上取得了较好的效果，比较显著的提升了生成图像的质量。因此本实验中，用实例归一化方法来替代批归一化方法。

$$y_{tijk} = \frac{x_{tijk} - \mu_{ti}}{\sqrt{\sigma_{ti}^2 + \epsilon}}, \quad \mu_{ti} = \frac{1}{HW} \sum_{l=1}^W \sum_{m=1}^H x_{tilm}, \quad \sigma_{ti}^2 = \frac{1}{HW} \sum_{l=1}^W \sum_{m=1}^H (x_{tilm} - \mu_{ti})^2 \quad (4.2)$$

• TensorFlow 中模型参数的恢复

在 TensorFlow 中，采用检查点机制（Checkpoint）周期地记录（Save）模型参数等数据并存储到文件系统中，后续当需要继续训练或直接使用训练好的参数做推断时，需要从文件系统中将保存的模型恢复（Restore）出来。检查点机制由 saver 对象来完成，即在模型训练过程中或当模型训练完成后，使用 saver=tf.train.Saver() 函数来保存模型中的所有变量。当需要恢复模型参数来继续训练模型或者进行预测时，需使用 saver 对象的 restore() 函数，从指定路径下的检查点文件中恢复出已保存的变量。在本实验中，图像转换网络和特征提取网络的参数均已经提前训练好并保存在特定路径下，在使用图像转换网络进行图像预测时，直接使用 restore() 函数将这些模型参数读入程序中并实现实时的风格迁移。

4.2.3 实验环境

本节实验所涉及的硬件平台和软件环境如下：

- 硬件平台：CPU、DLP
- 软件环境：TensorFlow 1.14，Python 编译环境及相关的扩展库，包括 Python 2.7.12，Pillow 4.2.1，Scipy 1.0.0，NumPy 1.16.6、CNML 高性能算子库、CNRT 运行时库

4.2.4 实验内容

由于基于 DLP 的实验平台已经提供了采用高性能库实现的 TensorFlow 框架，所以本节的主要实验内容是完成风格迁移模型在 DLP 定制版本 TensorFlow 上的运行，并和 CPU 版本的 TensorFlow 进行性能对比。具体实验内容包括：

1. 模型量化：由于 DLP 平台支持定点数据类型运算，为了提升模型处理的效率，先将原始的用 Float32 类型表示的 pb 模型文件量化为用 INT8 类型表示的模型文件。
2. 模型推断：将 pb 模型文件通过 Python 接口在 DLP 平台上运行推断过程，并和第 5 章在 CPU 上运行推断的性能进行对比。

第4.2和4.3节的代码参考自：[https://github.com/lengstrom/fast-style-transfer/。](https://github.com/lengstrom/fast-style-transfer/)

4.2.5 实验步骤

本实验主要包括以下步骤：读取图像、CPU 上实现、DLP 上实现、实验运行与对比等。

4.2.5.1 读取图像

使用与上一实验相同的方法读取一张图片，如果程序中指定了图片尺寸，就将该图像缩放至指定的尺寸。该部分代码定义在实验环境的 src/utils.py 文件中，如图 4.9 所示。

```

1 # file: src/utils.py
2 import scipy.misc
3 import numpy as np
4
5 def get_img(src, img_size = False):
6     #TODO: 使用scipy.misc模块读入输入图像src并转化成'RGB'模式,返回ndarray类型数组img
7     img = _____
8     _____
9
10    return img
11

```

图 4.9 读取输入图像

4.2.5.2 CPU 上实现实时风格迁移

为了在 CPU 上实现实时风格迁移，需要使用图像转换网络对应的 pb 模型文件处理输入图像，得到风格迁移后的输出图像。主要包括实时风格迁移函数和实时风格迁移主函数的定义等。

1. 实时风格迁移函数定义

以图 4.10 中的代码为例说明实时风格迁移函数的定义。该定义在 stu_upload/evaluate_cpu.py 文件中。

2. 实时风格迁移主函数定义

以图 4.11 的代码为例说明实时风格迁移主函数的定义。该定义同样在 stu_upload/evaluate_cpu.py 文件中。

3. 执行实时风格迁移

在 CPU 上运行如图 4.12 所示命令，实现图像的实时风格迁移。其中，模型文件 *.pb 保存在 pb_models/ 目录下，输入的内容图像保存在 data/train2014_small/ 目录下，风格迁移后的图像保存在 out/ 目录下。

4.2.5.3 DLP 上实现实时风格迁移

在 DLP 上实现实时风格迁移的实验步骤分为：模型量化和模型推断。

1. 模型量化

已经提前训练好的图像转换网络的数据类型为 Float32，需要经过量化后才可以在 DLP 上运行。在 fppb_to_intpb 目录下运行以下命令，使用量化工具完成对模型的量化，生成新模型 udnie_int8.pb。

```
! python fppb_to_intpb.py udnie_int8.ini
```

```
1 # file: evaluate_cpu.py
2 from __future__ import print_function
3 import sys
4 sys.path.insert(0, 'src')
5 import transform, NumPy as np, vgg, pdb, os
6 import scipy.misc
7 import tensorflow as tf
8 from utils import save_img, get_img, exists, list_files
9 from argparse import ArgumentParser
10 from collections import defaultdict
11 import time
12 import json
13 import subprocess
14 import numpy
15 BATCH_SIZE = 4
16 DEVICE = '/cpu:0'
17
18
19 os.putenv('MLU_VISIBLE_DEVICES', '')#设置MLU_VISIBLE_DEVICES="" 来屏蔽DLP
20
21 def ffwd(data_in, paths_out, model, device_t='/gpu:0', batch_size=1):
22     #该函数为风格迁移预测基础函数, data_in为输入的待转换图像, 它可以是保存了一张或多张输入图像的文件路径, 也可以是已经读入图像并转化成数组形式的数据; paths_out为存放输出图像的数组; model为pb模型参数的保存路径
23
24     assert len(paths_out) > 0
25     is_paths = type(data_in[0]) == str
26
27     #TODO: 如果 data_in 是保存输入图像的文件路径, 即 is_paths 为 True, 则读入第一张图像, 由于 pb 模型的输入维度为  $1 \times 256 \times 256 \times 3$ , 因此需将输入图像的形状调整为  $256 \times 256$ , 并传递给 img_shape; 如果 data_in 是已经读入图像并转化成数组形式的数据, 即 is_paths 为 False, 则直接获取图像的 shape 特征 img_shape
28
29
30     g = tf.Graph()
31     config = tf.ConfigProto(allow_soft_placement=True,
32                            inter_op_parallelism_threads=1,
33                            intra_op_parallelism_threads=1)
34     config.gpu_options.allow_growth = True
35     with g.as_default():
36         with tf.gfile.FastGFile(model, 'rb') as f:
37             graph_def = tf.GraphDef()
38             graph_def.ParseFromString(f.read())
39             tf.import_graph_def(graph_def, name='')
40
41         with tf.Session(config=config) as sess:
42             sess.run(tf.global_variables_initializer())
43             input_tensor = sess.graph.get_tensor_by_name('X_content:0')
44             output_tensor = sess.graph.get_tensor_by_name('add_37:0')
45             batch_size = 1
46             #TODO: 读入的输入图像的数据格式为 HWC, 还需要将其转换成 NHWC
47             batch_shape = _____
48             num_iters = int(len(paths_out)/batch_size)
49             for i in range(num_iters):
50                 #分批次对输入图像进行处理
51                 pos = i * batch_size
52                 curr_batch_out = paths_out[pos:pos+batch_size]
53
54                 #TODO: 如果 data_in 是保存输入图像的文件路径, 则依次将输入图像集合文件路径下的 batch_size 张图像读入数组 X; 如果 data_in 是已经读入图像并转化成数组形式的数据, 则将该数组传递给 X
55
56                 start = time.time()
57                 #TODO: 使用 sess.run 来计算 output_tensor
58                 _preds = _____
59                 end = time.time()
60                 for j, path_out in enumerate(curr_batch_out):
61                     #TODO: 在该批次下调用 utils.py 中的 save_img() 函数对所有风格迁移后的图片进行存储
62
```

```

1 # file: evaluate_cpu.py
2 def ffwd_to_img(in_path, out_path, model, device='/cpu:0'):
3     #该函数将上面的ffwd()函数用于图像的实时风格迁移
4     paths_in, paths_out = [in_path], [out_path]
5     ffwd(paths_in, paths_out, model, batch_size=1, device_t=device)
6
7 def main():
8     #实时风格迁移预测函数主体
9     #build_parser()与check_opts()用于解析输入指令，这两个函数的定义见evaluate_cpu.py文件
10    parser = build_parser()
11    opts = parser.parse_args()
12    check_opts(opts)
13
14    if not os.path.isdir(opts.in_path):
15        #如果输入的opts.in_path是已经读入图像并转化成数组形式的数据，则执行风格迁移预测
16        if os.path.exists(opts.out_path) and os.path.isdir(opts.out_path):
17            out_path = os.path.join(opts.out_path, os.path.basename(opts.in_path))
18        else:
19            out_path = opts.out_path
20
21    #TODO: 执行风格迁移预测，输入图像为opts.in_path，转换后的图像为out_path，模型文件路径为opts.model
22    -----
23    else:
24        #如果输入的opts.in_path是保存输入图像的文件路径，则对该路径下的图像依次实施风格迁移预测
25        #调用list_files函数读取opts.in_path路径下的输入图像，该函数定义见utils.py
26        files = list_files(opts.in_path)
27        full_in = [os.path.join(opts.in_path,x) for x in files]
28        full_out = [os.path.join(opts.out_path,x) for x in files]
29
30    #TODO: 执行风格迁移预测，输入图像的保存路径为full_in，转换后的图像为full_out，模型文件路径为opts.model
31    -----
32
33 if __name__ == '__main__':
34     main()
35

```

图 4.11 实时风格迁移训练主函数

```

1 python evaluate_cpu.py --model pb_models/udnie.pb --in-path data/
   train2014_small/ --out-path out/
2

```

图 4.12 执行实时风格迁移

2. 模型推断

通过 DLP 定制的 TensorFlow 版本（其中大部分风格迁移的算子都通过 DLP 的高性能库支持）完成风格迁移模型的前向推断。为了使上层用户不感知底层硬件的迁移，定制的 TensorFlow 维持了上层的 Python 接口，用户可以通过 session config 配置 DLP 运行的相关参数以及使用相关接口进行量化。具体的运行时配置信息如图 4.13 所示，可以设置运行的核数和使用的数据类型等信息。在配置完 DLP 硬件相关的参数后，推断时模型的算子可自动运行在 DLP 上。

```
1 # file: evaluate_mlu.py
2 import tensorflow as tf
3 ...
4 #配置环境变量，设置程序运行在DLP上
5 os.putenv('MLU_VISIBLE_DEVICES','0')
6 ...
7 #在生成session实例前，配置DLP参数
8 config = tf.ConfigProto(allow_soft_placement=True,
9                         inter_op_parallelism_threads=1,
10                        intra_op_parallelism_threads=1)
11 config.mlu_options.data_parallelism = 1
12 config.mlu_options.model_parallelism = 1
13 config.mlu_options.core_num = 1
14 config.mlu_options.precision = "int8"
15 config.mlu_options.save_offline_model = True
16 sess = tf.Session(config = config, graph = graph)
```

图 4.13 用 DLP 进行模型推断时配置的参数

在运行完成后，统计 sess.run() 前后的运行时间，并与在 CPU 上的运行时间进行对比。

4.2.5.4 实验运行

根据第4.2.5.1节~第4.2.5.3节的描述补全 evaluate_cpu.py、evaluate_mlu.py、utils.py，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_4_student 目录下。

```
1 # 登录云平台
2 ssh root@xxx.xxx.xxx.xxx -p XXXXX
3 # 进入 /opt/code_chap_4_student 目录
4 cd /opt/code_chap_4_student
5 # 初始化环境
6 cd env
7 source env.sh
8
```

2. 代码实现

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

```
1 # 进入实验目录
2 cd exp_4_2_fast_style_transfer_infer_student
```

```

3 # 补全 utils.py
4 vim src/utils.py
5 # 补全 cpu 实现代码
6 vim stu_upload/evaluate_cpu.py
7 # 补全 mlu 实现代码
8 vim stu_upload/evaluate_mlu.py
9

```

3. CPU 运行

```

# cpu 上运行
./run_cpu.sh

```

4. DLP 运行

```

1 # 对 pb 模型进行量化
2 cd fppb_to_intpb
3 python fppb_to_intpb.py udnie_int8.ini
4 # mlu 上运行
5 ./run_mlu.sh
6 # 运行完整实验
7 python main_exp_4_2.py
8

```

4.2.6 实验评估

本实验的评估标准设定如下：

- 60 分标准：在 CPU 平台上正确实现实时风格迁移的推断过程，给定输入图像、权重参数，可以实时计算并输出风格迁移后的图像，同时给出对图像进行实时风格迁移的时间。
- 100 分标准：在完成 60 分标准的基础上，在 DLP 平台上，给定输入图像、权重参数，能够实时输出风格迁移后的图像，同时给出 DLP 和 CPU 平台上实现实时风格迁移的时间对比。

4.2.7 实验思考

- 1) 对于给定的输入图像集合、权重参数，在不改变图像转换网络结构的前提下如何提升预测速度？
- 2) 在调用 TensorFlow 内置的卷积及转置卷积函数 `tf.nn.conv2d()`、`tf.nn.conv2d_transpose()` 时，边缘扩充方式分别选择”SAME” 或是”VALID”，对生成的图像结果有何影响？
- 3) 请采用性能剖析/监控等工具分析在 DLP 平台上进行推断的性能瓶颈。如何利用多核 DLP 架构提升整体的吞吐？

4.3 实时风格迁移的训练

4.3.1 实验目的

掌握如何使用 TensorFlow 实现实时风格迁移模型的训练。具体包括：

- 1) 掌握使用 TensorFlow 定义损失函数的方法；
- 2) 掌握使用 TensorFlow 存储网络模型的方法；
- 3) 以实时风格迁移算法为例，掌握使用 TensorFlow 进行神经网络训练的方法。

实验工作量：约 60 行代码，约需 8 个小时。

4.3.2 背景介绍

在第4.2小节中，介绍了使用 TensorFlow 实现实时风格迁移的推断。本小节进一步介绍如何使用 TensorFlow 来实现实时风格迁移的训练的相关背景。

除了第4.2小节中介绍的图像转换网络，实时风格迁移算法中还包含了一个特征提取网络，整个实时风格迁移算法的流程如图 4.14 所示。特征提取网络采用在 ImageNet 数据集上预训练好的 VGG16 网络结构^[21]，其接收内容图像、风格图像以及图像转换网络输出的生成图像作为输入，这些输入通过 VGG16 的不同层来计算损失函数，再通过迭代的训练图像转换网络的参数来优化该损失函数，最终实现对图像转换网络的训练。其中，损失函数由特征重建损失 L_{feat} 和风格重建损失 L_{style} 两部分组成：

$$L = \mathbb{E}_x [\lambda_1 L_{feat}(f_w(x), y_c) + \lambda_2 L_{style}(f_w(x), y_s)] \quad (4.3)$$

其中， λ_1 和 λ_2 是权重参数。特征重建损失用卷积输出的特征计算视觉损失：

$$L_{feat}^j(\hat{y}, y) = \frac{1}{C_j H_j W_j} \|\phi_j(\hat{y}) - \phi_j(y)\|_2^2 \quad (4.4)$$

其中， C_j 、 H_j 、 W_j 分别表示第 j 层卷积输出特征图的通道数、高度和宽度， $\phi(y)$ 是损失网络中第 j 层卷积输出的特征图，实际中选择第 7 层卷积的特征计算特征重建损失。而第 j 层卷积后的风格重建损失为输出图像和目标图像的格拉姆矩阵的差的 F-范数：

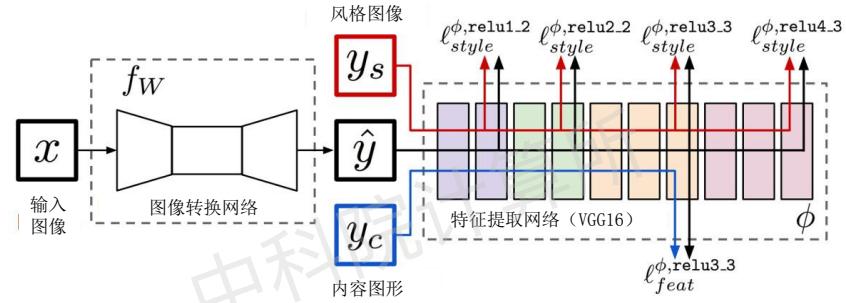
$$L_{style}^j(\hat{y}, y) = \|G_j(\hat{y}) - G_j(y)\|_F^2 \quad (4.5)$$

其中，格拉姆矩阵 $G_j(x)$ 为 $C_j \times C_j$ 大小的矩阵，矩阵元素为：

$$G_j(x)_{c,c'} = \frac{1}{C_j H_j W_j} \sum_{h=1}^{H_j} \sum_{w=1}^{W_j} \phi_j(x)_{h,w,c} \phi_j(x)_{h,w,c'} \quad (4.6)$$

风格重建损失为第 2、4、7、10 层卷积后的风格重建损失之和。

本实验中，为了平滑输出图像，消除图像生成过程中可能带来的伪影，在损失函数中增加了全变分正则化 (Total Variation Regularization)^[22] 部分。其计算方法为将图像水平和垂直方向各平移一个像素，分别与原图相减，然后计算两者 L^2 范数的和。此外，将特征提取网络的结构由 VGG16 替换成 VGG19，使得特征提取网络的网络深度更深，网络参数更多，这样网络的表达能力更强，特征提取的区分度更强，效果也更好。VGG19 的网络结构

图 4.14 实时图像风格迁移算法的流程^[15]表 4.4 VGG19 与 VGG16 的区别^[4]

配置					
A 11 个权重层	A-LRN 11 个权重层	B 13 个权重层	C 16 个权重层	D 16 个权重层	E 19 个权重层
输入 (224 × 224 大小的 RGB 图像)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
最大池化					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
最大池化					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
最大池化					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
最大池化					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
全连接层-4096					
全连接层-4096					
全连接层-1000					
Softmax					

与 VGG16 的区别如表 4.4 所示，表中的 D 列代表 VGG16 的网络配置，E 列代表 VGG19 的网络配置。

在训练图像转换网络的过程中，输入图像（即内容图像） \mathbf{x} 输入到图像转换网络进行处理，输出生成图像 $\hat{\mathbf{y}}$ ；再将生成图像 $\hat{\mathbf{y}}$ 、风格图像 \mathbf{y}_s 和内容图像 $\mathbf{y}_c = \mathbf{x}$ 分别送到特征提取网络中提取特征，并计算损失。

在使用 TensorFlow 进行实时风格迁移算法训练时，首先读入输入图像，构建特征提取网络，其构建方法和第 4.2 小节所采用的方法一致；然后定义损失函数，并创建优化器，定义模型训练方法；最后迭代地执行模型的训练过程。此外，在模型训练过程中或当模型训练完成后，可以使用 `tf.train.Saver()` 函数来创建一个 `saver` 实例，每训练一定次数就使用 `saver.save()` 函数将当前时刻的模型参数保存到磁盘指定路径下的检查点文件中。

4.3.3 实验环境

本节实验所涉及的硬件平台和软件环境如下：

- 硬件平台：CPU、DLP
- 软件环境：TensorFlow 1.14，Python 编译环境及相关的扩展库，包括 Python 2.7.12，Pillow 4.2.1，Scipy 1.0.0，NumPy 1.16.6、CNML 高性能算子库、CNRT 运行时库

4.3.4 实验内容

构建如图 4.14 所示的实时风格迁移网络，通过特征提取网络构建损失函数，并基于该损失函数来迭代的训练图像转换网络^[23]，最终获得较好的训练效果。

4.3.5 实验步骤

4.3.5.1 定义基本运算单元

如第 4.2 小节所述，实时风格迁移算法中的图像转换网络包含了卷积层、残差块、转置卷积层等几种不同的网络层。本部分需要分别定义出这几种不同网络层的计算方法。该部分代码定义在 `src/transform.py` 文件中。

1. 卷积层

以图 4.15 中的代码为例介绍图像转换网络中卷积层的定义方法。首先需要准备好权重的初值以及 `stride` 参数，然后进行卷积运算，并对计算结果进行批归一化处理和 ReLu 操作。

2. 残差块

以图 4.16 中的代码为例介绍图像转换网络中残差块的定义方法。根据 4.2.2 中介绍的残差块结构，利用上一步中实现好的卷积，实现残差块的功能。

3. 转置卷积层

以图 4.17 中的代码为例介绍图像转换网络中转置卷积层的定义方法。首先和卷积的实现一样，准备好权重的初值以及 `num_filters`、`strides` 参数，然后进行转置卷积计算，并对计算结果进行批归一化处理和 ReLu 操作。

```

1 # file: src/transform.py
2 import tensorflow as tf
3
4 def _conv_layer(net, num_filters, filter_size, strides, relu=True):
5     #该函数定义了卷积层的计算方法，net为该卷积层的输入 ndarray 数组，num_filters 表示输出通道数，filter_size 表示卷积核尺寸，strides 表示卷积步长，该函数最后返回卷积层计算的结果
6
7     #TODO: 准备好权重的初值
8     weights_init = _____
9
10    #TODO: 输入的 strides 参数为标量，需将其处理成卷积函数能够使用的数据形式
11    _____
12
13    #TODO: 进行卷积计算
14    net = _____
15
16    #TODO: 对卷积计算结果进行批归一化处理
17    net = _____
18
19    if relu:
20        #TODO: 对归一化结果进行 ReLU 操作
21        net = _____
22
23    return net
24

```

图 4.15 卷积层的定义

```

1 # file:src/transform.py
2 def _residual_block(net, filter_size=3):
3     #该函数定义了残差块的计算方法，net为该层的输入 ndarray 数组，filter_size 表示卷积核尺寸，该函数最后返回残差块的计算结果
4
5     #TODO: 调用上一步骤中实现的卷积层函数，实现残差块的计算
6     _____
7
8     return net
9

```

图 4.16 残差块的定义

```

1 # file:src/transform.py
2 def _conv_tranpose_layer(net, num_filters, filter_size, strides):
3     #该函数定义了转置卷积层的计算方法, net为该层的输入 ndarray 数组, num_filters 表示输出通道数,
4     #filter_size 表示卷积核尺寸, strides 表示卷积步长, 该函数最后返回转置卷积层计算的结果
5
6     #TODO: 准备好权重的初值
7     weights_init = _____
8
9     -----
10
11     #TODO: 输入的 num_filters、strides 参数为标量, 需将其处理成转置卷积函数能够使用的数据形式
12     -----
13
14     #TODO: 进行转置卷积计算
15     net = _____
16
17     #TODO: 对卷积计算结果进行批归一化处理
18     net = _____
19
20     #TODO: 对归一化结果进行 ReLU 操作
21     net = _____
22
23     return net

```

图 4.17 转置卷积层的定义

4.3.5.2 创建图像转换网络模型

在分别完成了卷积层、残差块、转置卷积层的定义以后，本步骤构建起如图 4.6 所示的图像转换网络模型。该部分代码定义在 `src/transform.py` 文件中，下面以图 4.18 中的代码为例展开介绍。图像转换网络的结构在 4.2.2 小节中已经介绍过，如图 4.6 所示，三个卷积层、五个残差块、两个转置卷积层再接一个卷积层构成。使用上面步骤实现好的卷积、残差块、转置卷积等基本运算单元，搭建图像转换网络，每一层的输出作为下一层的输入，并将最后一层的输出经过 `tanh` 函数处理，得到输出结果 `preds`。

```

1 # file: src/transform.py
2 def net(image):
3     #该函数构建图像转换网络, image 为步骤1中读入的图像 ndarray 阵列, 返回最后一层的输出结果
4
5     #TODO: 构建图像转换网络, 每一层的输出作为下一层的输入
6     conv1 = _____
7     conv2 = _____
8
9     -----
10
11     #TODO: 最后一个卷积层的输出再经过 tanh 函数处理, 最后的输出张量 preds 像素值需限定在 [0,255] 范围内
12     preds = _____
13
14     return preds

```

图 4.18 创建图像转换网络模型

4.3.5.3 定义特征提取网络

特征提取网络采用与第3.1节相同的VGG19模型文件，使用与第4.1小节类似的定义方法。图??是定义特征提取网络的程序代码，根据VGG19的网络结构，使用实现好的卷积等基本运算单元搭建VGG19网络。特征提取网络的参数使用官方的预训练模型，可以直接加载提供的imagenet-vgg-verydeep-19.mat文件。该部分代码定义在src/vgg.py文件中。

```

1 # file: src/vgg.py
2 import tensorflow as tf
3 import numpy as np
4 import scipy.io
5 import pdb
6
7 def net(data_path, input_image):
8     # 定义特征提取网络，data_path为网络参数的保存路径，input_image为已经通过get_img()函数读取并转换成ndarray格式的内容图像
9
10    # TODO: 根据VGG19的网络结构定义每一层的名称
11    layers = (
12        'conv1_1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1',
13        -----
14    )
15
16    # TODO: 从data_path路径下的.mat文件中读入已训练好的特征提取网络参数weights
17    -----
18
19    net = {}
20    current = input_image
21    for i, name in enumerate(layers):
22        kind = name[:4]
23        if kind == 'conv':
24            # TODO: 如果当前层为卷积层，则进行卷积计算，计算结果为current
25            -----
26        elif kind == 'relu':
27            # TODO: 如果当前层为ReLU层，则进行ReLU计算，计算结果为current
28            -----
29        elif kind == 'pool':
30            # TODO: 如果当前层为池化层，则进行最大池化计算，计算结果为current
31            -----
32        net[name] = current
33
34    assert len(net) == len(layers)
35    return net

```

图4.19 定义特征提取网络

4.3.5.4 损失函数构建

输入图像（即内容图像）通过图像转换网络输出生成图像；再将生成图像、风格图像、内容图像分别送到特征提取网络的特定层中提取特征，并计算损失。损失函数由特征重建损失 $content_loss$ 、风格重建损失 $style_loss$ 和全变分正则化项 tv_loss 组成。损失函数构建的程序示例如下所示。该部分代码定义在src/optimize.py文件中，同时会调用前面步骤中实现的vgg.py及transform.py文件。

```
1 # file: src/optimize.py
```

```

2 from __future__ import print_function
3 import functools
4 import vgg, pdb, time
5 import tensorflow as tf, NumPy as np, os
6 import transform
7 from utils import get_img
8
9 STYLE_LAYERS = ('relu1_1', 'relu2_1', 'relu3_1', 'relu4_1', 'relu5_1')
10 CONTENT_LAYER = 'relu4_2'
11 DEVICES = '/CPU:0'
12
13 def _tensor_size(tensor):
14     #对张量进行切片操作，将NHWC格式的张量，切片成HWC，再计算H、W、C的乘积
15     from operator import mul
16     return functools.reduce(mul, (d.value for d in tensor.get_shape()[1:]), 1)
17
18 def loss_function(net, content_features, style_features, content_weight, style_weight, tv_weight,
19     preds, batch_size):
20     #损失函数构建，net为特征提取网络，content_features为内容图像特征，style_features为风格图像特征，content_weight、style_weight和tv_weight分别为特征重建损失、风格重建损失的权重和全变分正则化损失的权重
21
22     batch_shape = (batch_size, 256, 256, 3)
23
24     #TODO: 计算内容损失
25     content_size = _tensor_size(content_features[CONTENT_LAYER])*batch_size
26     assert _tensor_size(content_features[CONTENT_LAYER]) == _tensor_size(net[CONTENT_LAYER])
27     content_loss = _____
28
29     #计算风格损失
30     style_losses = []
31     for style_layer in STYLE_LAYERS:
32         layer = net[style_layer]
33         bs, height, width, filters = map(lambda i:i.value,layer.get_shape())
34         size = height * width * filters
35         feats = tf.reshape(layer, (bs, height * width, filters))
36         feats_T = tf.transpose(feats, perm=[0,2,1])
37         grams = tf.matmul(feats_T, feats) / size
38         style_gram = style_features[style_layer]
39         #TODO: 计算style_losses
40
41         _____
42         style_loss = style_weight * functools.reduce(tf.add, style_losses) / batch_size
43
44         #使用全变分正则化方法定义损失函数tv_loss
45         tv_y_size = _tensor_size(preds[:,1:,:,:])
46         tv_x_size = _tensor_size(preds[:, :, 1:,:])
47         #TODO: 将图像preds向水平和垂直方向各平移一个像素，分别与原图相减，分别计算二者的L2范数x_tv和y_tv
48         _____
49         tv_loss = tv_weight*2*(x_tv/tv_x_size + y_tv/tv_y_size)/batch_size
50
51     loss = content_loss + style_loss + tv_loss
52     return content_loss, style_loss, tv_loss, loss

```

4.3.5.5 实时风格迁移训练的实现

在完成了特征提取网络及损失函数的定义后，接下来需要完成实时风格迁移的训练部分，主要包括优化器的创建、实时风格迁移训练方法和主函数的定义等。该部分代码定义在src/optimize.py以及style.py文件中，同时会调用前几个步骤中实现的vgg.py、transform.py、

utils.py 以及 evaluate.py 文件。

1. 实时风格迁移训练方法定义

以下面的代码来说明实时风格迁移训练方法的定义。该函数定义在 src/optimize.py 文件中，同时会调用前面步骤中实现的 vgg.py、transform.py 以及 utils.py 文件。

```

1 # file: optimize.py
2 from __future__ import print_function
3 import functools
4 import vgg, pdb, time
5 import tensorflow as tf, NumPy as np, os
6 import transform
7 from utils import get_img
8
9 STYLE_LAYERS = ('relu1_1', 'relu2_1', 'relu3_1', 'relu4_1', 'relu5_1')
10 CONTENT_LAYER = 'relu4_2'
11 DEVICES = '/CPU:0'
12
13 def optimize(content_targets, style_target, content_weight, style_weight,
14             tv_weight, vgg_path, epochs=2, print_iterations=1000,
15             batch_size=4, save_path='saver/fns.ckpt', slow=False,
16             learning_rate=1e-3, debug=True):
17     #实时风格迁移训练方法定义, content_targets为内容图像, style_target为风格图像, content_weight、
18     #style_weight和tv_weight分别为特征重建损失、风格重建损失和全变分正则化项的权重, vgg_path为保存
19     #VGG19网络参数的文件路径
20     if slow:
21         batch_size = 1
22     mod = len(content_targets) % batch_size
23     if mod > 0:
24         print("Train set has been trimmed slightly..")
25         content_targets = content_targets[:-mod]
26
27     #风格特征预处理
28     style_features = {}
29     batch_shape = (batch_size, 256, 256, 3)
30     style_shape = (1,) + style_target.shape
31     print(style_shape)
32
33     with tf.Graph().as_default(), tf.device(DEVICES), tf.Session() as sess:
34         #使用NumPy库在CPU上处理
35
36         #TODO: 使用占位符来定义风格图像 style_image
37         style_image = _____
38
39         #TODO: 依次调用 vgg.py 文件中的 preprocess()、net() 函数对风格图像进行预处理，并将此时得到的特征提取网络传递给 net
40         _____
41
42         #使用NumPy库对风格图像进行预处理, 定义风格图像的格拉姆矩阵
43         style_pre = np.array([style_target])
44         for layer in STYLE_LAYERS:
45             features = net[layer].eval(feed_dict={style_image:style_pre})
46             features = np.reshape(features, (-1, features.shape[3]))
47             gram = np.matmul(features.T, features) / features.size
48             style_features[layer] = gram
49
50         #TODO: 先使用占位符来定义内容图像 X_content, 再调用 preprocess() 函数对 X_content 进行预处理, 生成 X_pre
51         _____
52
53         #提取内容特征对应的网络层
54         content_features = {}
55         content_net = vgg.net(vgg_path, X_pre)
56         content_features[CONTENT_LAYER] = content_net[CONTENT_LAYER]
```

```
55
56     if slow:
57         preds = tf.Variable(tf.random_normal(X_content.get_shape()) * 0.256)
58         preds_pre = preds
59     else:
60         #TODO: 内容图像经过图像转换网络后输出结果 preds，并调用 preprocess() 函数对 preds 进行预处理，生成 preds_pre
61         -----
62
63     #TODO: preds_pre 输入到特征提取网络，并将此时得到的特征提取网络传递给 net
64     net = -----
65
66     #TODO: 计算内容损失 content_loss, 风格损失 style_loss, 全变分正则化项 tv_loss, 损失函数 loss
67     -----
68     #TODO: 创建 Adam 优化器，并定义模型训练方法为最小化损失函数方法，返回 train_step
69     -----
70     #TODO: 初始化所有变量
71     -----
72     import random
73     uid = random.randint(1, 100)
74     print("UID: %s" % uid)
75     save_id = 0
76     for epoch in range(epochs):
77         num_examples = len(content_targets)
78         iterations = 0
79         while iterations * batch_size < num_examples:
80             start_time = time.time()
81             curr = iterations * batch_size
82             step = curr + batch_size
83             X_batch = np.zeros(batch_shape, dtype=np.float32)
84             for j, img_p in enumerate(content_targets[curr:step]):
85                 X_batch[j] = get_img(img_p, (256,256,3)).astype(np.float32)
86
87             iterations += 1
88             assert X_batch.shape[0] == batch_size
89
90             feed_dict = {
91                 X_content:X_batch
92             }
93
94             train_step.run(feed_dict=feed_dict)
95             end_time = time.time()
96             delta_time = end_time - start_time
97             is_print_iter = int(iterations) % print_iterations == 0
98             if slow:
99                 is_print_iter = epoch % print_iterations == 0
100            is_last = epoch == epochs - 1 and iterations * batch_size >= num_examples
101            should_print = is_print_iter or is_last
102            if should_print:
103                to_get = [style_loss, content_loss, loss, preds]
104                test_feed_dict = {
105                    X_content:X_batch
106                }
107
108                tup = sess.run(to_get, feed_dict = test_feed_dict)
109                _style_loss, _content_loss, _loss, _preds = tup
110                losses = (_style_loss, _content_loss, _loss)
111                if slow:
112                    _preds = vgg.unprocess(_preds)
113                else:
114                    with tf.device('/CPU:0'):
115                        #TODO: 将模型参数保存到 save_path，并将训练的次数 save_id 作为后缀加入到模型名字中
116                        -----
```

```

117     #将相关计算结果返回
118     yield(_preds, losses, iterations, epoch)

```

2. 实时风格迁移训练主函数

以下面的代码来说明实时风格迁移训练的主函数。该函数定义在 style.py 文件中，同时会调用前面步骤中实现的 optimize.py、utils.py 以及 evaluate.py 文件^①。

```

1 # file: style.py
2 from __future__ import print_function
3 import sys, os, pdb
4 sys.path.insert(0, 'src')
5 import numpy as np, scipy.misc
6 from optimize import optimize
7 from argparse import ArgumentParser
8 from utils import save_img, get_img, exists, list_files
9 import evaluate
10
11 os.putenv('MLU_VISIBLE_DEVICES', '')#设置MLU_VISIBLE_DEVICES="" 来屏蔽DLP
12 CONTENT_WEIGHT = 7.5e0
13 STYLE_WEIGHT = 1e2
14 TV_WEIGHT = 2e2
15
16 LEARNING_RATE = 1e-3
17 NUM_EPOCHS = 2
18 CHECKPOINT_DIR = 'checkpoints'
19 CHECKPOINT_ITERATIONS = 2000
20 VGG_PATH = 'data/imagenet-vgg-verydeep-16.mat'
21 TRAIN_PATH = 'data/train2014'
22 BATCH_SIZE = 4
23 DEVICE = '/cpu:0'
24 FRAC_GPU = 1
25
26 def _get_files(img_dir):
27     #读入内容图像目录下的所有图像并返回
28     files = list_files(img_dir)
29     return [os.path.join(img_dir,x) for x in files]
30
31 def main():
32     #build_parser()与check_opts()用于解析输入指令，这两个函数的定义见style.py文件
33     parser = build_parser()
34     options = parser.parse_args()
35     check_opts(options)
36
37     #TODO: 获取风格图像 style_target 以及内容图像数组 content_targets
38     -----
39
40     if not options.slow:
41         content_targets = _get_files(options.train_path)
42     elif options.test:
43         content_targets = [options.test]
44
45     kwargs = {
46         "epochs":options.epochs,
47         "print_iterations":options.checkpoint_iterations,
48         "batch_size":options.batch_size,
49         "save_path":os.path.join(options.checkpoint_dir, 'fns.ckpt'),

```

^①受 CPU 硬件算力的限制，完整跑完整个训练流程可能需要花费较多时间，因此，本实验中可以仅跑完前几百个 iterations，同时每隔 100 个 iterations 即打印计算出的 loss 值，观察 loss 值随着训练的进行逐步减小的过程。

```

50     "learning_rate":options.learning_rate
51 }
52
53 if options.slow:
54     if options.epochs < 10:
55         kwargs['epochs'] = 1000
56     if options.learning_rate < 1:
57         kwargs['learning_rate'] = 1e1
58
59 args = [
60     content_targets,
61     style_target,
62     options.content_weight,
63     options.style_weight,
64     options.vgg_path
65 ]
66
67 for preds, losses, i, epoch in optimize(*args, **kwargs):
68     style_loss, content_loss, tv_loss, loss = losses
69
70     print('Epoch %d, Iteration: %d, Loss: %s' % (epoch, i, loss))
71     to_print = (style_loss, content_loss, tv_loss)
72     print('style: %s, content:%s, tv: %s' % to_print)
73
74 ckpt_dir = options.checkpoint_dir
75 print("Training complete.\n")
76
77 if __name__ == '__main__':
78     main()
79

```

3. 执行实时风格迁移的训练

在实验环境中运行命令，实现实时风格迁移算法的训练。其中，生成的模型文件 *.ckpt 保存在 ckp_temp/路径下，输入的风格图像保存在 examples/style/路径下^①。

```

1 python style.py --checkpoint-dir ckp_temp \
2   --style examples/style/rain_princess.jpg \
3   --train-path data/train2014_small \
4   --content-weight 1.5e1 \
5   --checkpoint-iterations 100 \
6   --epochs 2 \
7   --batch-size 4 \
8   --type 0
9

```

4.3.5.6 实验运行

根据第4.3.5.1节～第4.3.5.5节的描述补全 optimize.py、transform.py、utils.py、vgg.py、style.py，并通过 Python 运行.py 代码。具体可以参考以下步骤。

^①在进行训练之前，建议首先使用以下语句以检查数据集是否完好：

```
find . -name jpg -exec identify -verbose -regard-warnings >/dev/null "+"
```

该语句依赖 identify 命令，如当前环境不支持，可以使用“apt-get install imagemagick”命令来安装相应依赖库。

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_4_student目录下。

```

1 # 登录云平台
2 ssh root@xxx.xxx.xxx.xxx -pxxxxx
3 # 进入 /opt/code_chap_4_student 目录
4 cd /opt/code_chap_4_student
5 # 初始化环境
6 cd env
7 source env.sh
8

```

2. 代码实现

补全 src 目录中的 optimize.py、transform.py、utils.py、vgg.py 文件。

```

1 # 进入实验目录
2 cd exp_4_3_fast_style_transfer_train_student
3 # 补全 utils.py
4 vim src/utils.py
5 # 补全 transform.py
6 vim src/transform.py
7 # 补全 vgg.py
8 vim src/vgg.py
9 # 补全 optimize.py
10 vim src/optimize.py
11 # 补全训练主函数 style.py
12 vim style.py
13

```

3. 运行实验

```

1 # 运行完整实验
2 python main_exp_4_3.py
3 # 单独进行训练过程
4 ./run_style.sh
5

```

4.3.6 实验评估

本实验的评估标准设定如下:

- **60 分标准:** 正确实现特征提取网络及损失函数的构建。给定输入的内容图像、风格图像,首先通过图像转换网络输出生成图像,再根据内容图像、生成图像以及风格图像来计算损失函数值。正确实现实时风格迁移的训练过程,给定输入图像、风格图像,可以通过训练过程使得损失值逐渐减少。
- **80 分标准:** 在图像转换网络中使用实例归一化替代批归一化,正确实现实时风格迁移的训练过程,给定输入图像、风格图像,可以通过训练过程使得损失值逐渐减少。

- 100 分标准：正确实现检查点文件的保存及恢复功能，使得每经过一定训练迭代次数即将当前参数保存在特定检查点文件中，且图像转换网络可使用该参数生成图像，以验证训练效果。

4.3.7 实验思考

- 1) 整个实时风格迁移算法中包含了图像转换网络和特征提取网络两部分，其中特征提取网络的参数是已经预训练好的，在使用 TensorFlow 设计算法时，应该如何操作才能使得训练时 TensorFlow 内置的优化器仅针对图像转换网络的参数进行优化？
- 2) 对于给定的输入图像集合，在不改变图像转换网络以及特征提取网络结构的前提下应如何提升训练速度？
- 3) 在图像转换网络中使用实例归一化方法，相比批归一化方法，对生成的图像质量会产生怎样的影响？
- 4) 为什么计算风格损失时需要将多层卷积层的输出求和，而计算内容损失时只需要计算第四层卷积层的输出？
- 5) 在定义损失函数时，如果改变内容损失、风格损失和全变分正则化损失权重 *content_weight*, *style_weight* 和 *tv_weight*，将对最后的迁移效果起到怎样的作用？

4.4 自定义 TensorFlow CPU 算子

4.4.1 实验目的

掌握如何在 TensorFlow 中新增自定义的 PowerDifference 算子。具体包括：

- 1) 熟悉 TensorFlow 整体设计机理；
- 2) 通过对风格迁移 pb 模型的扩展，掌握对 TensorFlow pb 模型进行修改的方法，理解 TensorFlow 如何以计算图的方式完成对深度学习算法的处理；
- 3) 通过在 TensorFlow 框架中添加自定义的 PowerDifference 算子，加深对 TensorFlow 算子实现机制的理解，掌握在 TensorFlow 中添加自定义 CPU 算子的能力，为后续在 TensorFlow 中集成添加自定义的 DLP 算子奠定基础。

实验工作量：约 40 行代码，约需 4 个小时。

4.4.2 背景介绍

4.4.2.1 PowerDifference 介绍

实时风格迁移的训练和预测过程中，实例归一化和损失计算均需要用 SquaredDifference 计算均方误差。本实验将 SquaredDifference 算子扩展替换成更通用的 PowerDifference 算子，用于对两个张量的差值进行次幂运算。其具体计算公式如下：

$$\text{PowerDifference} = (\mathbf{X} - \mathbf{Y})^Z \quad (4.7)$$

其中 **X** 和 **Y** 是张量数据类型，**Z** 是标量数据类型。由于张量 **X** 和 **Y** 的形状 (shape) 可能不一致，有可能无法直接进行按元素的减法操作，因此 PowerDifference 的计算通常需要

三个步骤：首先将输入 \mathbf{X} 和 \mathbf{Y} 进行数据广播操作，统一形状后做减法，然后再进行求幂运算。与原始风格迁移模型中的 SquaredDifference 算子（完成 $(\mathbf{X} - \mathbf{Y})^2$ 运算）相比，自定义的 PowerDifference 算子具有更好的通用性。

4.4.2.2 添加 TensorFlow 算子流程

本节介绍在 TensorFlow 框架中添加自定义算子的主要流程。

首先，最简便的方式是根据已有的 Python 操作（Op）来构造新的算子。其次，如果采用 Python 的方式难以构造或者无法满足要求时，可以考虑采用底层 C++ 来实现新的算子。具体来说，是否采用 C++ 来实现自定义算子有以下几条基本准则：

1. 使用现有的 Op 并加以组合成新的算子不易实现或无法实现；
2. 将新的算子表示为现有 Op 的组合无法达到预期效果；
3. 开发者期望自由融合一些 Op，但编译器很难将其融合。

例如，算法设计人员在设计模型时想实现中值池化（Median Pooling）算子。该算子类似于最大池化，但是在进行滑动窗口操作时使用中位数来代替最大值。我们可以使用一系列 Op 的拼接来完成该功能，如 ExtractImagePatches（用以提取图片中特定区域）和 TopK 算子，但是其会存在性能问题或造成不必要的内存浪费。

如果基于底层 C++ 来实现并添加新的自定义算子，一般需要以下 5 个步骤：

1. 用 C++ 实现算子的具体功能。算子的实现称为 Kernel。针对不同的输入、输出类型或硬件架构（如 CPU、GPU 或 DLP 等），可以有多个 Kernel 实现。
2. 在 C++ 文件中注册新算子。算子的注册与其具体实现是相互独立的，在注册时定义的接口主要为了描述该算子如何执行。例如，算子注册函数定义了其名字、输入和输出，还可定义用于推断张量形状的函数。
3. 创建 Python 封装器（可选）。该封装器用于在 Python 中创建此算子的公共 API。默认的封装器是通过算子注册并遵循特定规则生成的，用户可以直接使用。
4. 编写该算子的梯度计算函数（可选）。
5. 编译测试。对 TensorFlow 进行重新编译并进行测试。通常在 Python 中测试该操作，开发人员也可以在 C++ 中进行操作测试。如果定义了梯度，可以使用 Python 的 GradientChecker 来进行测试。

4.4.3 实验环境

本节实验所涉及的硬件平台和软件环境如下：

- 硬件平台：CPU
- 软件环境：TensorFlow 1.14，bazel 0.24.1，Python 编译环境及相关的扩展库，包括 Python 2.7.12，Pillow 4.2.1，Scipy 1.0.0，NumPy 1.16.6、CNML 高性能算子库、CNRT 运行时库

4.4.4 实验内容

基于第4.3节训练得到的风格迁移模型，我们将其中的 `SquaredDifference` 算子替换成为更通用的 `PowerDifference` 算子。由于原始 TensorFlow 框架中并不支持该算子，直观的解决方案是采用 Python 的 NumPy 扩展包实现该算子。与基于 NumPy 的实现相比，如果可以直接受扩展 TensorFlow 的算子库，使 TensorFlow 框架直接支持该算子有望大幅提升处理效率。为了体现基于 Python 的实现和基于 TensorFlow 框架实现的区别，本节所设计的实验内容如图 4.20 所示。主要流程包括：

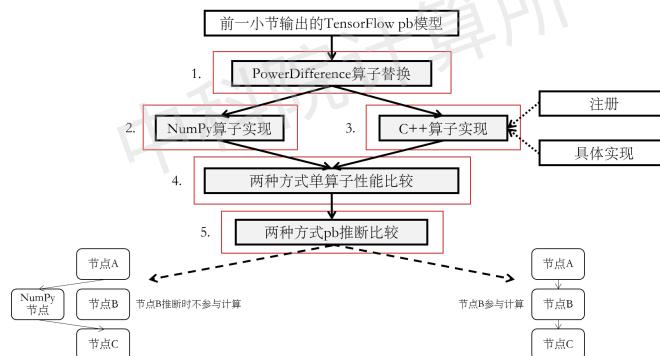


图 4.20 自定义算子实验流程图

1. 模型扩展：TensorFlow pb 模型节点的修改与扩展；
2. NumPy 实现：使用 NumPy 实现该算子的计算过程；
3. C++ 实现：使用 C++ 完成该算子的 CPU 实现并集成在 TensorFlow 框架中；
4. 算子测试：编写测试用例，比较使用 NumPy 和 C++ 不同方式实现算子的性能差异；
5. 模型测试：比较使用上述两种不同方式执行整个 pb 模型推断的性能差异。

最后一个步骤中包含 NumPy 算子和 C++ 算子的 pb 模型推断测试。对于 NumPy 算子需要将 `PowerDifference` 算子的输入直接传递给 CPU，使用第二步中完成的 NumPy 函数进行计算，然后将结果作为该节点的下一层输入，统计使用该方法的推理时间。对于 C++ 算子，仅需正常执行推断程序，统计 TensorFlow 中 `Session.run()` 前后的时间即可。

4.4.5 实验步骤

如前所述，完整的实验流程分为：模型扩展、NumPy 实现、C++ 实现、算子测试及模型测试等 5 部分。

4.4.5.1 模型扩展

模型扩展的主要目的是增加输入节点使得 `PowerDifference` 算子中的幂指数可以顺利传入 pb 模型中，同时模型中被替换的 `SquaredDifference` 节点名称要相应进行修改。具体包括以下步骤：

1. 转换模型文件

采用如图 4.21 所示的 `pb2pbtxt` 工具，将 pb 模型转换为可读的 pbtxt 格式。以 `udnie.pb` 模型为例，通过以下命令生成 `udnie.pbtxt` 文件：

```
1 python pb_to_pbtxt.py models/pb_models/udnie.pb udnie.pbtxt.
```

```
1 # file: pb_to_pbtxt.py
2 import argparse
3 import tensorflow as tf
4 from tensorflow.core.framework import graph_pb2
5
6 if __name__ == '__main__':
7     parser = argparse.ArgumentParser()
8     parser.add_argument('input_pb', help='input pb to be converted')
9     parser.add_argument('output_pbtxt', help='output pbtxt generated')
10    args = parser.parse_args()
11    with tf.Session() as sess:
12        with tf.gfile.FastGFile(args.input_pb, 'rb') as f:
13            graph_def = graph_pb2.GraphDef()
14            graph_def.ParseFromString(f.read())
15            #tf.import_graph_def(graph_def)
16            tf.train.write_graph(graph_def, '.', args.output_pbtxt, as_text=True)
```

图 4.21 pb-pbtxt 转换工具

2. 添加输入节点:

编辑生成的 udnie.pbtxt 文件，在首行添加如图 4.22 所示的节点信息。

```
1 node {
2     name: "moments_15/PowerDifference_z"
3     op: "Placeholder"
4     attr {
5         key: "dtype"
6         value {
7             type: DT_FLOAT
8         }
9     }
10    attr {
11        key: "shape"
12        value {
13            shape {
14                unknown_rank: true
15            }
16        }
17    }
18}
```

图 4.22 pbtxt 添加新输入节点

3. 修改节点名

找到模型文件中的最后一个 SquaredDifference 节点，将其修改为 PowerDifference 节点，如图 4.23 所示。注意，除了修改最后一个 SquaredDifference 节点，还需要将其它以该节点作为输入的节点（此处为“moments_15/variance”）的“input”域统一从 SquaredDifference 替换为 PowerDifference，如图 4.24 所示。

4. 输出扩展模型：采用如图 4.25 所示的 pbtxt2pb 工具，将编辑后的 udnie.pbtxt 输出为扩展后的 pb 模型 udnie_power_diff.pb。

```

1 node {
2   name: "moments_15 / PowerDifference"
3   op: "PowerDifference"
4   input: "Conv2D_13"
5   input: "moments_15 / StopGradient"
6   input: "moments_15 / PowerDifference_z"
7   attr {
8     key: "T"
9     value {
10       type: DT_FLOAT
11     }
12   }
13 }
```

图 4.23 pbtxt 修改节点属性

```

1 node {
2   name: "moments_15 / variance"
3   op: "Mean"
4   input: "moments_15 / PowerDifference"
5   input: "moments_15 / variance / reduction_indices"
6   attr {
7     key: "T"
8     value {
9       type: DT_FLOAT
10    }
11  }
12  attr {
13    key: "Tidx"
14    value {
15      type: DT_INT32
16    }
17  }
18  attr {
19    key: "keep_dims"
20    value {
21      b: true
22    }
23  }
24 }
```

图 4.24 pbtxt 修改输入节点

4.4.5.2 NumPy 实现

NumPy 实现主要指根据 PowerDifference 的原理，使用 Python 的 NumPy 扩展包实现其数学计算。NumPy 实现的程序如下所示：

```

1 # file: power_diff_Numpy.py
2 import numpy as np
3 def power_diff_Numpy(input_x, input_y, input_z):
4     #Reshape 操作，根据实时风格迁移模型的实际情况，该函数假设input_x和input_y的最后一个维度的dim
5     #size 相同，input_y除了最后的维度，其余dim size 均为1。
6     x_shape = np.shape(input_x)
7     y_shape = np.shape(input_y)
8     x = np.reshape(input_x, (-1, y_shape[-1]))
9     x_new_shape = np.shape(x)
10    y = np.reshape(input_y, (-1))
11    output = []
12    #TODO: 通过 for 循环完成计算，每次循环计算 y 个数的 PowerDifference 操作
```

```

1 #file: pbtxt_to_pb.py
2 import argparse
3 import tensorflow as tf
4 from tensorflow.core.framework import graph_pb2
5 from google.protobuf import text_format
6
7 if __name__ == '__main__':
8     parser = argparse.ArgumentParser()
9     parser.add_argument('input_pbtxt', help='input pbtxt to be converted')
10    parser.add_argument('output_pb', help='output pb generated')
11    args = parser.parse_args()
12    with tf.Session() as sess:
13        with tf.gfile.FastGFile(args.input_pbtxt, 'rb') as f:
14            graph_def = graph_pb2.GraphDef()
15            new_graph_def = text_format.Merge(f.read(), graph_def)
16            tf.train.write_graph(new_graph_def, '.', args.output_pb, as_text=False)

```

图 4.25 pbtxt-pb 转换工具

```

12 for i in range(x_new_shape[0]):
13     -----
14     -----
15 return output

```

4.4.5.3 C++ 实现

C++ 实现主要指在 TensorFlow 框架中集成用 C++ 编写的算子，以进行高效的模型推断。下面基于第 4.4.2.2 节中关于 TensorFlow 算子添加流程的背景知识，从算子实现、算子注册、算子编译三方面展开详细介绍。

1. 算子实现

PowerDifference 的实现定义在 tensorflow/core/kernels/cwise_op_power_difference.cc 文件中，其部分代码如下所示：

```

1 # file: cwise_op_power_difference.cc
2 template <typename T>
3 class PowerDifferenceOp : public OpKernel {
4 public:
5     explicit PowerDifferenceOp(OpKernelConstruction* context)
6         : OpKernel(context) {}
7
8     void Compute(OpKernelContext* context) override {
9         const Tensor& input_x_tensor = context->input(0);
10        const Tensor& input_y_tensor = context->input(1);
11        const Tensor& input_pow_tensor = context->input(2);
12
13        const Eigen::ThreadPoolDevice& device = context->eigen_device<Eigen::ThreadPoolDevice>();
14        // BCast 部分，调用了 TensorFlow 自有的 BCast 算子，确保 x 和 y 的 shape 一致
15        BCast bcast(BCast::FromShape(input_y_tensor.shape()), BCast::FromShape(input_x_tensor.shape()));
16        /* fewer_dims_optimization= */ true);
17
18        Tensor* output_tensor = nullptr;
19        TensorShape output_shape = BCast::ToShape(bcast.output_shape());
20

```

```

21    OP_REQUIRES_OK(context,
22                    context->allocate_output(0, output_shape, &output_tensor));
23
24    Tensor input_x_broad(input_x_tensor.dtype(), output_shape);
25    Tensor input_y_broad(input_y_tensor.dtype(), output_shape);
26
27    .....
28
29    auto input_x = input_x_broad.flat<T>();
30    auto input_y = input_y_broad.flat<T>();
31    auto input_pow = input_pow_tensor.flat<T>();
32    auto output = output_tensor->flat<T>();
33
34    const int N = input_x.size();
35    const int POW = input_pow(0);
36    float tmp = 0;
37    // 实际计算部分
38    for (int i = 0; i < N; i++) {
39        // output(i) = (input_x(i) - input_y(i)) * (input_x(i) - input_y(i));
40        tmp = input_x(i) - input_y(i);
41        output(i) = tmp;
42        for (int j = 0; j < POW - 1; j++) {
43            output(i) = output(i) * tmp;
44        }
45    }
46}
47};

```

其中主要包括 CPU 实现算子的构造函数 PowerDifferenceOp 和 Compute 方法。在 Compute 方法中，首先需调用 TensorFlow 已有的 BCast 算子来实现对 Tensor 的广播操作，使得输入的两个张量 input_x 和 input_y 的形状一致，最后同样用循环的方式完成所有元素的计算。

2. 算子注册

首先在 tensorflow/core/ops 目录下找到对应的算子注册文件。TensorFlow 对于算子注册位置没有特定的限制，为了尽可能和算子的用途保持一致，此处选择注册常用数学函数的 math_ops.cc 文件。在该文件下添加如图 4.26 所示的信息。然后在文件 tensorflow/core/kernels/cwise_op_power_difference.cc 文件中添加如图 4.27 所示的信息。

```

1 REGISTER_OP("PowerDifference")
2     .Input("x: T")
3     .Input("y: T")
4     .Input("pow: T")
5     .Output("z: T")
6     .Attr(
7         "T: {bfloat16, float, half, double, int32, int64, complex64, "
8         "complex128}")
9     .SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
10         c->set_output(0, c->input(0));
11         c->set_output(0, c->input(1));
12         c->set_output(0, c->input(2));
13         return Status::OK();
14     });

```

图 4.26 TensorFlow 注册 (1)

```

1 REGISTER_KERNEL_BUILDER( \\
2   Name("PowerDifference").Device(DEVICE_CPU), \\
3   PowerDifferenceOp<float>);

```

图 4.27 TensorFlow 注册 (2)

3. 算子编译

注册完成后，需进一步将此文件添加至 core/kernel 的 BUILD 文件，使其能够正常编译。注意，如果在算子实现中用到了 TensorFlow 中已有的算子（例如在 PowerDifference 的实现中用到了 BCast 算子），也需要将对应依赖添加至 BUILD 文件中，具体代码如图 4.28 所示。该步骤完成后可以采用如图 4.29 所示的命令重新编译 TensorFlow 源码，完成编译后，即可使用 Python 进行该 API 的调用。

```

1 filegroup( \\
2   name = "android_extended_ops_group1", \\
3   srcs = [ \\
4     "argmax_op.cc", \\
5     ..... \\
6     "cwise_op_power_difference.cc", \\
7     // 剩余注册文件 \\
8   ], \\
9 ) \\
10 ..... \\
11 tf_kernel_library( \\
12   name = "cwise_op", \\
13   prefix = "cwise_op", \\
14   deps = MATH_DEPS, \\
15   hdrs = [ \\
16     "broadcast_to_op.h" \\
17   ] + ..... , \\
18 )

```

图 4.28 修改 BUILD 文件

```

1 #1. 执行指令，查看主机bazel的版本信息，版本号需大于等于0.24，如不满足需先进行相应的升级。 \\
2 bazel version \\
3 #2. 激活环境 \\
4 source env.sh \\
5 #3. 执行编译脚本 \\
6 cd /path/to/tensorflow/source \\
7 build_tensorflow-v1.10_mlu.sh

```

图 4.29 TensorFlow 的编译

4.4.5.4 算子测试

算子测试指的是通过编写测试程序，测试采用 NumPy 与 C++ 实现算子的精度与性能。在测试程序文件夹下创建 data 文件夹存放测试数据，包含“in_x.txt”，“in_y.txt”，“in_z.txt”以及正确结果“out.txt”四个文档。其中，in_x.txt、in_y.txt 和 in_z.txt 三个文档储存 PowerDifference 的三个输入，out.txt 文档储存 PowerDifference 测试用例的正确输出结果。示例测试程序如下所示：

```

1 # file: power_difference_test_cpu.py
2 import numpy as np
3 import os
4 import time
5 import tensorflow as tf
6 from power_diff_Numpy import *
7 np.set_printoptions(suppress=True)
8
9 def power_difference_op(input_x ,input_y ,input_pow):
10     with tf.Session() as sess:
11         # TODO: 完成TensorFlow 接口调用
12
13         return sess.run(out , feed_dict = {...})
14
15 def main():
16     start = time.time()
17     input_x = np.loadtxt("./data/in_x.txt",delimiter=',')
18     input_y = np.loadtxt("./data/in_y.txt")
19     input_pow = np.loadtxt("./data/in_z.txt") #some specific number
20     output = np.loadtxt("./data/out.txt")
21     end = time.time()
22     print("load data cost "+ str((end-start)*1000) + "ms" )
23     # test C++ PowerDifference CPU Op
24     start = time.time()
25     res = power_difference_op(input_x , input_y , input_pow)
26     end = time.time()
27     print("comput C++ op cost "+ str((end-start)*1000) + "ms" )
28     err = sum(abs(res - output))/sum(output)
29     print("C++ op err rate= "+ str(err*100))
30     # test NumPy PowerDifference Op
31     start = time.time()
32     res = power_diff_Numpy(input_x , input_y , input_pow)
33     end = time.time()
34     print("comput NumPy op cost "+ str((end-start)*1000) + "ms" )
35     err = sum(abs(res - output))/sum(output)
36     print("NumPy op err rate= "+ str(err*100))
37 if __name__ == '__main__':
38     main()

```

4.4.5.5 模型测试

模型测试指的是分别采用 NumPy 和 C++ 实现的算子进行网络模型推断测试。

1. 基于 NumPy 算子进行模型推断

首先需要修改 TensorFlow 的 pb 模型, 将 PowerDifference 节点删除并增加一个相同名字的输入节点, 使得 NumPy 计算后的数据可以传入到 pb 模型中, 其步骤可以参考第 4.4.5.1 小节中介绍的方法和流程。在 pbtxt 中添加 NumPy 计算节点的信息如下所示:

```

1 node {
2     name: "moments_15 / PowerDifference"
3     op: "Placeholder"
4     attr {
5         key: "dtype"
6         value {
7             type: DT_FLOAT
8         }
9     }

```

```

10   attr {
11     key: "shape"
12     value {
13       shape {
14         unknown_rank: true
15       }
16     }
17   }
18 }
```

得到新模型后使用 NumPy 完成缺失节点的计算，NumPy 实现的程序如下所示：

```

1 # file: transform_cpu.py
2 def run_NumPy_pb():
3     args = parse_arg()
4     config = tf.ConfigProto(allow_soft_placement=True,
5                            inter_op_parallelism_threads=1,
6                            intra_op_parallelism_threads=1)
7     model_name = os.path.basename(args.NumPy_pb).split('.')[0]
8     image_name = os.path.basename(args.image).split('.')[0]
9
10    g = tf.Graph()
11    with g.as_default():
12        with tf.gfile.FastGFile(args.NumPy_pb, 'rb') as f:
13            graph_def = tf.GraphDef()
14            graph_def.ParseFromString(f.read())
15            tf.import_graph_def(graph_def, name='.')
16        img = cv.imread(args.image)
17        X = cv.resize(img, (256, 256))
18        with tf.Session(config=config) as sess:
19            sess.graph.as_default()
20            sess.run(tf.global_variables_initializer())
21
22        # TODO: 根据输入名称获得输入的 tensor
23        input_tensor = _____
24        # TODO: 获取两个输出节点, 作为 NumPy 算子的输入
25        out_tmp_tensor_1 = _____
26        out_tmp_tensor_2 = _____
27        # TODO: 执行第一次 session run, 得到 NumPy 算子的两个输入值, 注意此时两个输入的 shape 不同
28        input_x, input_y = _____
29        input_pow = 2 # 幂指数参数, 可设为其它值, 此处设置为2
30        output = power_diff_Numpy(input_x, input_y, input_pow)
31
32        # TODO: 根据添加的输入节点名称获得输入 tensor
33        input_tensor_new = _____
34        # TODO: 完整推断最终输出的 tensor
35        output_tensor = _____
36        # TODO: 执行第二次 session run, 输入图片数据以及上一步骤 NumPy 计算的数据
37        ret = _____
38        # 结果保存
39        img1 = tf.reshape(ret,[256,256,3])
40        img_Numpy = img1.eval(session=sess)
41        cv.imwrite('result_new.jpg',img_Numpy)
```

NumPy 计算完缺失的节点信息后，需将计算结果即图中的 output 作为输入数据传入 pb 进行完整计算。

2. 基于 C++ 算子进行模型推断

采用 C++ 进行模型推断的代码如下所示：

```
1 # file: transform_cpu.py
2 def run_ori_power_diff_pb(ori_power_diff_pb, image):
3     config = tf.ConfigProto(allow_soft_placement=True,
4                            inter_op_parallelism_threads=1,
5                            intra_op_parallelism_threads=1)
6     model_name = os.path.basename(ori_power_diff_pb).split('.')[0]
7     image_name = os.path.basename(image).split('.')[0]
8
9     g = tf.Graph()
10    with g.as_default():
11        with tf.gfile.FastGFile(ori_power_diff_pb, 'rb') as f:
12            graph_def = tf.GraphDef()
13            graph_def.ParseFromString(f.read())
14            tf.import_graph_def(graph_def, name='')
15        img = cv.imread(image)
16        X = cv.resize(img, (256, 256))
17        with tf.Session(config=config) as sess:
18            # TODO: 完成 PowerDifference pb 模型的推断
19            -----
20
21            start_time = time.time()
22            ret = sess.run(...)
23            end_time = time.time()
24            print("C++ inference(CPU) time is: ", end_time - start_time)
25            img1 = tf.reshape(ret, [256, 256, 3])
26            img_Numpy = img1.eval(session=sess)
27            cv.imwrite(image_name + '_'+ model_name + '_cpu.jpg', img_Numpy)
```

由于此时 PowerDifference 已集成到了框架内部，因此对于模型的推断仅需调用一次 Session.run() 即可完成。

4.4.5.6 实验运行

根据第4.4.5.1节~第4.4.5.5节的描述补全 cwise_op_power_difference.cc，并将其集成到TensorFlow中，然后补全 power_diff_Numpy.py、power_difference_test_cpu.py、transform_cpu.py等文件，并通过Python运行上述代码。

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_4_student目录下。

```
1 # 登录云平台
2 ssh root@xxx.xxx.xxx.xxx -p xxxxx
3 # 进入 /opt/code_chap_4_student 目录
4 cd /opt/code_chap_4_student
5 # 初始化环境
6 cd env
7 source env.sh
8
```

2. 代码实现

补全 src 目录中的 optimize.py、transform.py、utils.py、vgg.py 文件。

```

1 # 进入实验目录
2 cd exp_4_4_custom_tensorflow_op_student
3 # 修改模型
4 python pb_to_pbtxt.py models/pb_models/udnie.pb udnie.pbtxt
5 vim udnie.pbtxt
6 # c++ 算子模型
7 python pbtxt_to_pb.py udnie.pbtxt udnie_power_diff.pb
8 # numpy 算子模型
9 python pbtxt_to_pb.py udnie.pbtxt udnie_power_diff_numpy.pb
10 # 补全 power_diff_numpy.py
11 vim stu_upload/power_diff_numpy.py
12 # 补全 cwise_op_power_difference.cc
13 vim tf-implementation/cwise_op_power_difference.cc
14 # 算子集成
15 cp tf-implementation/cwise_op_power_difference.* /opt/code_chap_4_student/env/tensorflow-
v1.10/tensorflow/core/kernels/
16 vim /opt/code_chap_4_student/env/tensorflow-v1.10/tensorflow/core/kernels/BUILD
17 # 编译 tensorflow
18 /opt/code_chap_4_student/env/tensorflow-v1.10/build_tensorflow-v1.10_mlu.sh
19 # 补全 power_difference_test_cpu.py
20 vim stu_upload/power_difference_test_cpu.py
21 # 补全 transform_cpu.py
22 vim stu_upload/transform_cpu.py
23

```

3. 运行实验

```

1 # 将 tensorflow_mlu-1.14.0-cp27-cp27mu-linux_x86_64.whl, udnie_power_diff.pb 和
2 udnie_power_diff_numpy.pb 复制到 stu_upload 目录下
3 cp /opt/code_chap_4_student/env/tensorflow-v1.10/virtualenv_mlu/tensorflow_mlu-1.14.0-cp27-
cp27mu-linux_x86_64.whl stu_upload/
4 cp udnie_power_diff.pb stu_upload/
5 cp udnie_power_diff_numpy stu_upload/
6 # 运行完整实验
7 ./run_exp_4_4.py

```

4.4.6 实验评估

本实验的评估标准设定如下：

- 60 分标准：完成 PowerDifference NumPy 算子和 C++ 算子的编写和注册工作，对于实验平台提供的测试数据可以做到精度正常。
- 80 分标准：在 60 分基础上，对于实验平台提供的大规模测试数据（多种输入 Shape）可以做到精度正常。
- 100 分标准：在 80 分基础上，基于两种算子实现方式进行模型推断的精度正常，且 C++ 实现方式性能优于 NumPy 实现方式。

4.4.7 实验思考

- 1) 为何不用 NumPy 来实现算子的编写？框架相比 NumPy 实现来说有什么好处？

- 2) 框架内部核心代码为何使用 C/C++ 语言, 为何不使用高级语言(如 Python 等)?
- 3) 使用 Python 的框架接口和使用 C++ 框架接口有何不同?
- 4) 不同深度学习框架之间有何联系与区别?

第5章 智能编程语言

智能编程语言是连接智能编程框架和智能计算硬件的桥梁。本章将通过具体实验阐述智能编程语言的开发，优化和集成技巧。

具体而言，第5.1节介绍如何用智能编程语言BCL实现用户自定义的高性能库算子（即PowerDifference），并将其集成到TensorFlow框架中。希望读者可以通过本实验了解如何将BCL与智能编程框架集成，以将前述训练好的风格迁移网络模型编译为DLP硬件指令，满足智能计算系统的可扩展和高性能需求。第5.2节介绍BCL的一些高级特性，使读者在实现功能的基础上进一步掌握性能优化充分发挥DLP硬件潜力的编程技巧。

5.1 智能编程语言算子开发与集成实验

5.1.1 实验目的

本实验通过智能编程语言实现PowerDifference算子，掌握使用智能编程语言进行算子开发，扩展高性能库算子，并最终集成到TensorFlow框架中的方法和流程，使得完整的风格迁移网络可以在DLP硬件上高效执行。

实验工作量：代码量约150行，实验时间约10小时。

5.1.2 背景介绍

本节重点介绍面向智能编程语言开发所需的编译工具链，包括编译器、调试器及集成开发环境等。

5.1.2.1 编译器（CNCC）

CNCC是将使用智能编程语言（BCL）编写的程序编译成DLP底层指令的编译器。为了填补高层智能编程语言和底层DLP硬件指令间的鸿沟，DLP的编译器通过复杂寄存器分配、自动软件流水、全局指令调度等技术实现编译优化，以提升生成的二进制指令性能。

CNCC的结构如图5.1所示，开发者使用BCL开发自己的DLP端的源代码：首先通过前端CNCC编译为汇编代码，然后汇编代码由CNAS汇编器生成DLP上运行的二进制机器码。

在使用CNCC编译BCL文件时，有多个编译选项供开发者使用，如表5.1所示。

5.1.2.2 调试器（CNGDB）

CNGDB是面向智能编程语言所编写程序的调试工具，能够支持搭载DLP硬件的异构平台调试，即同时支持Host端C/C++代码和Device端BCL的调试，同时两者调试过程的切换对于用户而言也是透明的。此外，针对多核DLP架构的特点，调试器能同时支持单核

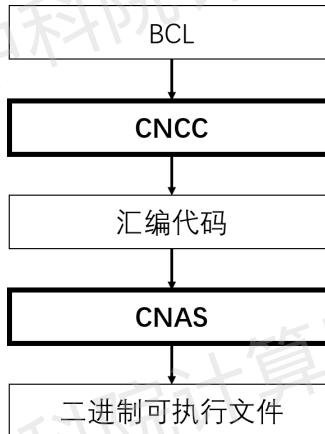


图 5.1 CNCC

表 5.1 CNCC 编译选项

常用选项	说明
-E	编译器只执行预处理的步骤，生成预处理文件
-S	编译器只执行预处理、编译的步骤，生成汇编文件
-c	编译器只执行预处理、编译、汇编的步骤，生成 ELF 格式的汇编文件
-o	将输出写入到指定的文件
-x	为输入的文件指定编程语言，如 BCL 等
-target=	指定生成 DLP 端的目标文件的格式，该文件和目标 Host 平台的目标文件一起链接成目标 Host 端的可执行程序，所以其值为目标 Host 端二进制文件格式，eg x86_64, armv7a 等
--bang-mlu-arch=	为输入的 BCL 程序指定 DLP 的架构
--bang-stack-on-l dram	栈是否放在 LDRAM 上，默认放在 NRAM 上，如果该选项开启，栈会放在 LDRAM 上
--cnas-path=	指定汇编器的路径

和多核应用程序的调试。CNGDB 解决了异构编程模型调试的问题，提升了应用程序开发的效率。

为了使用 CNGDB 进行调试，在使用 CNCC 编译 BCL 文件时，需要使用-g 选项，在-O0 优化级别中来获取含有调试信息的二进制文件，例如图 5.2 中的命令：

```
cncc kernel.mlu -o kernel.o --bang-mlu-arch=MLU270 -g -O0
```

图 5.2 使用 CNCC 编译一个 BCL 文件

这里以 BCL 写的快速排序程序为例，演示如何使用 CNGDB 调试程序。图 5.3 展示了 CNGDB 调试 recursion.mlu 程序的基本流程，总的来说主要包含如下几个步骤：断点插入、程序执行、变量打印、单步调试和多核切换等。

5.1.2.3 集成开发环境（CNStudio）

CNStudio 是一款针对于 BCL 语言可在 Visual Studio Code 使用的编程插件，为了使 BCL 语言在编写过程中更加方便快捷，CNStudio 基于 VSCode 编译器强大的功能和简便的

```

1 #1. 在 CNCC 编译时开启 -g 选项，首先将 recursion.mlu 文件编译为带有调试信息的二进制文件：
2 cncc recursion.mlu -o recursion.o --bang=mlu--arch=MLU270 -g -O0
3 #2. 然后继续编译得到可运行二进制文件：
4 g++ recursion.o main.cpp -o quick_sort -lcrt -I${DLP_INC} -L${DLP_LIB}
5 #3. 在有DLP板卡的机器上使用CNGDB打开quick_sort程序：
6 cngdb quick_sort
7 #4. 用 break 命令，在第x行添加断点：
8 (cn-gdb) b recursion.mlu :x
9 #5. 用 run 命令，执行程序至断点处，此时程序执行至 kernel 函数的 x 行处(x 行还未执行)
10 (cn-gdb) r
11 #6. 用 print 命令，分别查看第一次调用 x 行函数时的三个实参：
12 (cn-gdb) p input1
13 (cn-gdb) p input2
14 (cn-gdb) p input3
15 #7(a). 如果使用 continue 命令，程序会从当前断点处继续执行直到结束。如果不希望程序结束，可以
    继续添加断点：
16 (cn-gdb) c
17 #7(b). 如果希望进入被调用的某函数内部，可以直接使用 step 命令，达到单步调试的效果：
18 (cn-gdb) s
19 #8. 可以使用 info args 命令和 info locals 命令查看函数参数以及函数局部变量：
20 (cn-gdb) info args
21 #9. 如果需要对不同核进行调试，通过切换焦点获取对应 core 的控制权：
22 (cngdb) cngdb focus Device 0 cluster 0 core 2

```

图 5.3 CNGDB 调试示例

可视化操作提供包括语法高亮、自动补全和程序调试等功能。

当前 CNStudio 插件只提供离线安装包，不支持在线安装，安装包的具体下载地址请参考本课程网站。其中，VSCode 版本要求 1.28.0 及以上版本。具体安装流程如图 5.4 所示。通过上述步骤找到对应的离线安装包，完成 CNStudio 插件的安装。



图 5.4 CNStudio 安装流程图

安装完毕后，在左侧插件安装界面的搜索框中输入“@installed”即可查询全部插件，若显示如图 5.5 所示的插件则说明 CNStudio 安装成功。注意：CNStudio 的高亮颜色与 VScode 背景颜色会有冲突，可通过组合快捷键 (Ctrl+k) (Ctrl+t) 更改浅色主题。

在创建工程时（以新建一个 DLP 文件夹为例），由于每个 project 都包含三种类型的文件，需要在 DLP 文件夹中新建 DLP 端程序所需的 dlp.mlu（Device 端程序源文件后缀名为“*.mlu”）。安装 CNStudio 插件后，vscode 会自动识别 mlu 文件），Host 端程序所需的 main.cpp，以及头文件 kernel.h。可通过 VScode 工具栏中“File”→“Save Workspace As...”，将

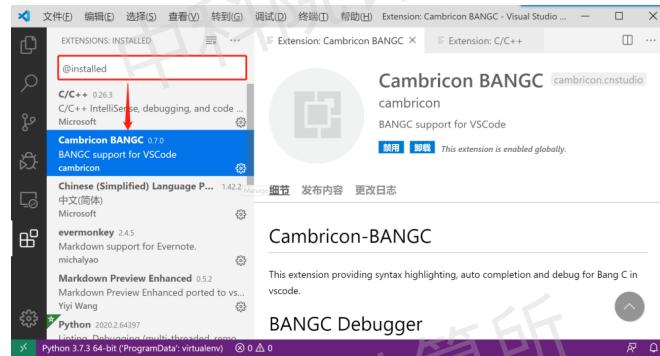


图 5.5 CNStudio 安装完成

打开的 DLP 工程保存为 workspace，方便下次直接打开工程文件。

5.1.2.4 BCL 算子库 (CNPlugin)

CNPlugin 是一款包含了一系列 BCL 算子的高性能计算库。通过 CNPlugin 算子库机制可以帮助 BCL，CNML 与框架之间协同工作，有机融合。CNPlugin 在 CNML 层提供一个接口，将 BCL 语言生成的算子与 CNML 的执行逻辑统一起来。

为了将 BCL kernel 函数与 CNML 结合运行，如图5.6所示，CNML 提供了一套相关 API 来达到这个目的，通过这种 API 运行的算子被称为 PluginOp。

```

1 CNML_DLL_API cnmlStatus_t cnmlCreatePluginOp(cnmlBaseOp_t *op,
2                                                 const char *name,
3                                                 void *kernel,
4                                                 cnrtKernelParamsBuffer_t params,
5                                                 cnmlTensor_t *input_tensors,
6                                                 int input_num,
7                                                 cnmlTensor_t *output_tensors,
8                                                 int output_num,
9                                                 cnmlTensor_t *statics_tensor,
10                                            int static_num);
11
12 CNML_DLL_API cnmlStatus_t cnmlComputePluginOpForward_V4(cnmlBaseOp_t op,
13                                                       cnmlTensor_t input_tensors[],
14                                                       void *inputs[],
15                                                       int input_num,
16                                                       cnmlTensor_t output_tensors[],
17                                                       void *outputs[],
18                                                       int output_num,
19                                                       cnrtQueue_t queue,
20                                                       void *extra);

```

图 5.6 CNML PluginOp 相关的主要 API

在以上 CNML 的基础之上，如图5.7所示，CNPlugin 中的每个算子都包含了 Host 端代码 (plugin_yolov3_detection_output_op.cc 文件) 和 Device 端 BCL 代码 (plugin_yolov3_detection_output_kernel 文件)。其中 Host 端代码主要完成了算子参数处理，CNML PluginOp API 封装和 BCL Kernel 调用等工作。Device 端代码包含了 BCL 源码，实现了主要的计算逻辑。

在 CNPlugin 中如果函数参数不多可以选择直接传参的方式，如果参数比较多则建议

```

1 Cambricon-CNPlugin-MLU270
2 └── build
3     ├── build_aarch64.sh
4     ├── build_cnplugin.sh
5     ├── CMakeLists.txt
6     └── common
7         └── include
8             └── cnplugin.h
9     └── pluginops
10    └── PluginYolov3DetectionOutputOp
11        ├── plugin_yolov3_detection_output_kernel_v2.h
12        ├── plugin_yolov3_detection_output_kernel_v2.mlu
13        └── plugin_yolov3_detection_output_op.cc
14     └── README.md
15     └── samplecode

```

图 5.7 CNPlugin 的主要目录结构示意

OpParam 结构体来完成传参。具体包括了结构体内容, CreatePluginOpParam 函数和 cnmlDestroyPluginOpParam 函数。如图5.8是一个 Addpad 算子的 OpParam 示例。

```

1 struct cnmlPluginAddpadOpParam {
2     int batch_size;
3     int src_h;
4     int src_w;
5     int dst_h;
6     int dst_w;
7     int type_uint8;
8     int type_yuv;
9 };
10
11 cnmlStatus_t cnmlCreatePluginAddpadOpParam(cnmlPluginAddpadOpParam_t *param_ptr,
12                                              int batch_size ,
13                                              int src_h ,
14                                              int src_w ,
15                                              int dst_h ,
16                                              int dst_w ,
17                                              int type_uint8 ,
18                                              int type_yuv) {
19     *param_ptr = new cnmlPluginAddpadOpParam();
20     (*param_ptr)->batch_size = batch_size;
21     (*param_ptr)->src_h = src_h;
22     (*param_ptr)->src_w = src_w;
23     (*param_ptr)->dst_h = dst_h;
24     (*param_ptr)->dst_w = dst_w;
25     (*param_ptr)->type_uint8 = type_uint8;
26     (*param_ptr)->type_yuv = type_yuv;
27     return CNML_STATUS_SUCCESS;
28 }
29
30 cnmlStatus_t cnmlDestroyPluginAddpadOpParam(cnmlPluginAddpadOpParam_t param) {
31     delete param;
32     return CNML_STATUS_SUCCESS;
33 }

```

图 5.8 CNPlugin OpParam 示例

5.1.3 实验环境

本节实验所涉及的硬件平台和软件环境如下：

- 硬件平台：硬件平台基于前述的 DLP 云平台环境。
- 软件环境：所涉及的 DLP 软件开发模块包括编程框架 TensorFlow、高性能库 CNML、运行时库 CNRT、编程语言及编译器。

5.1.4 实验内容

本节实验基于第4.4节中的高性能库算子实验，在前者基础上进一步把 PowerDifference 算子用智能编程语言实现，通过高性能库 PluginOp 接口扩展算子，并和高性能库原有算子一起集成到编程框架 TensorFlow 中，此后将风格迁移模型在扩展后的 TensorFlow 上运行，最后将其性能结果和第4.4节中的性能结果进行对比。实验内容和流程如图5.9所示，主要包括：

1. 算子实现：采用智能编程语言 BCL 实现 PowerDifference 算子并完成相应测试。首先，使用 BCL 的内置向量函数实现计算 Kernel，并利用 CNRT 接口直接调用 Kernel 运行并测试功能正确性；
2. 框架集成：通过高性能库 PluginOp 的接口对 PowerDifference 算子进行封装，使其调用方式和高性能库原有算子一致，将封装后的算子集成到 TensorFlow 框架中并进行测试，保证其精度和性能正确；
3. 在线推理：通过 TensorFlow 框架的接口，在内部高性能库 CNML 和运行时库 CNRT 的配合下，完成对风格迁移模型的在线推理，并生成离线模型；
4. 离线推理：采用运行时库 CNRT 的接口编写应用程序，完成离线推理，并将其结果和第三步中的在线推理，以及第4.4节中的推理性能进行对比。

图5.9中虚线框的部分是需要同学补充完善的实验文件。每一步实验操作需要修改的具体对应文件内容请参考下一节“实验步骤”。

5.1.5 实验步骤

如前所述，本实验的详细步骤包括：算子实现、框架集成、在线推理和离线推理等。

5.1.5.1 算子实现

为了实现 PowerDifference 算子，需要完成 Kernel 程序编写、运行时程序编写、Main 程序编写和编译运行等步骤。

1. Kernel 程序编写 (`plugin_power_difference_kernel.mlu`)

本节实验的第一步是使用 BCL 实现 PowerDifference 算子。具体的算子公式请参考 4.4.2.1 小节的内容。实验的主要内容需要完成 `_mlu_entry_` 函数供 CNRT 或 CNML 调用。这样可供调用的 `_mlu_entry_` 函数称为一个 Kernel。基于智能编程语言的 PowerDifference (`plugin_power_difference_kernel.mlu`) 具体实现如图 5.10 代码所示：

在上述代码中，为了充分发挥算子性能，使用了向量计算函数来完成运算。为了使用向量计算函数必须要满足两个前提。第一是调用计算时数据的输入和输出存放位置必须在

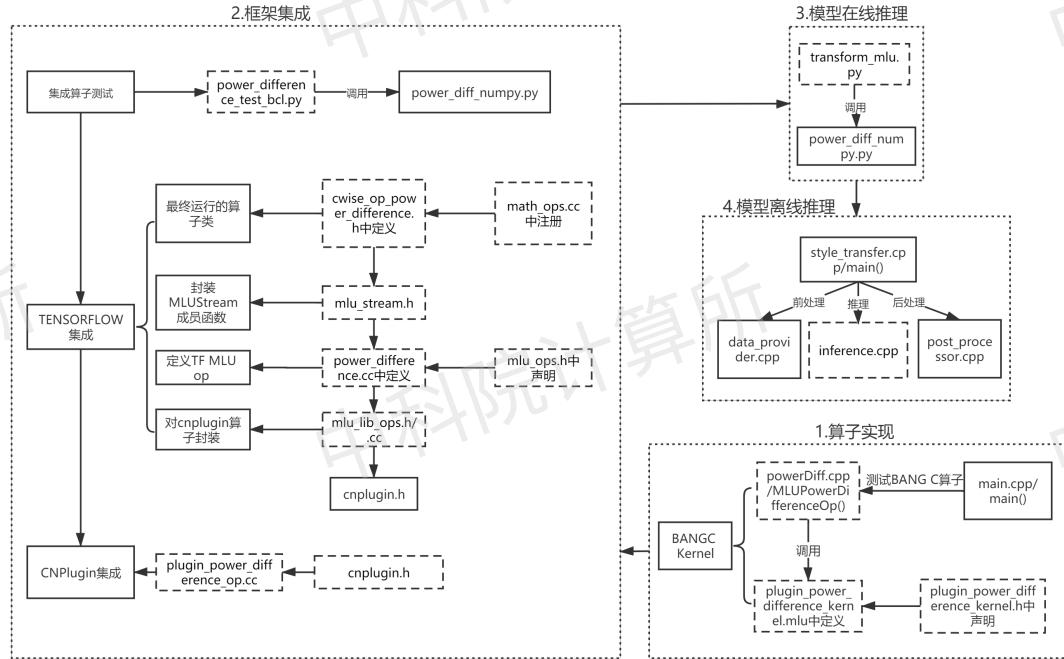


图 5.9 具体实验内容

NRAM 上。这要求我们必须在计算前先使用 `memcpy` 将数据从 GDRAM 拷贝到 NRAM 上。在计算完成后也要将结果从 NRAM 拷贝到 GDRAM 上。第二是向量操作的输入规模必须对齐到 64 的整数倍。在这里程序将数据对齐到 256。

由于 NRAM 大小的限制，不能一次性将所有数据全部拷贝到 NRAM 上执行，因此需要对原输入规模进行分块。这里分块的规模在满足 NRAM 大小和函数对齐要求的前提下由用户指定，这里设置为 256 (ONELINE)。分块的重点在于余数段的处理。由于通常情况下输入不一定是 256 的倍数，所以最后会有一部分长度小于 256，大于 0 的余数段。读者在完成实验时需注意该部分数据的处理逻辑。

2. 运行时程序编写 (powerDiff.cpp)

运行时程序通过利用运行时库 CNRT 的接口调用 BCL 算子来实现。如图 5.11 所示，首先声明被调用的算子实现函数，然后在 `MLUPowerDifferenceOp` 中通过一系列 CNRT 接口的调用完成，包括：使用 `cnrtKernelParamsBuffer` 来设置 `PowerDifference` 算子的输入参数，通过 `cnrtInvokeKernel` 来调用算子 Kernel 函数 (`PowerDifferenceKernel`)，最后完成计算后获取输出结果并销毁相应资源。对应的程序代码如下所示：

3. Main 程序编写 (main.cpp)

Main 程序首先读取文件 `in_x.txt` 和 `in_y.txt` 中的数据加载到内存中，然后调用上一步定义的 `MLUPowerDifferenceOp` 函数对输入数据进行计算，并将结果输出到文件 `out.txt` 中。其中会统计计算时间，并得到和 CPU 运算结果相对比的错误率。具体代码如 5.12 所示：

4. 编译运行 (power_diff_test)

完成上述代码的编写后，需要编译运行该程序。具体的编译命令如图 5.13 所示。其中，

```

1 // filename: plugin_power_difference_kernel.mlu
2 // 定义常量
3 #define ONELINE 256
4
5 __mlu_entry__ void PowerDifferenceKernel(half* input1, half* input2, int pow, half*
6     output, int len)
7 {
8     int quotient = len / ONELINE;
9     __bang_printf("%d %d\n", pow, len);
10    int rem = len % ONELINE;
11
12    // 声明NRAM空间
13    __nram__ half input1_nram[ONELINE];
14    __nram__ half input2_nram[ONELINE];
15
16    if (rem != 0)
17    {
18        quotient +=1;
19    }
20    for (int i = 0; i < quotient; i++)
21    {
22        // 内存拷贝：从GDRAM的 (input1 + i * ONELINE) 位置开始，拷贝ONELINE * sizeof(half)大
23        // 小的数据到input1_nram空间中。
24        __memcpy(_____);
25        __memcpy(_____);
26        // 按元素减法操作，将input1_nram和input2_nram的对应元素进行相减并储存在input1_nram
27        // 中。
28        __bang_sub(_____);
29        // NRAM中两个数据块的数据拷贝操作
30        __memcpy(_____);
31        for (int j = 0; j < pow - 1; j++)
32        {
33            // 按元素相乘操作
34            __bang_mul(_____);
35        }
36        // 内存拷贝：从NRAM中将计算的结果拷出至GDRAM中。
37        __memcpy(_____);
38    }
39 }

```

图 5.10 基于智能编程语言 BCL 的 Power Difference 实现

首先调用编译器 CNCC 将算子实现函数编译成为 powerdiffkernel.o 文件，然后通过 Host 的 g++ 编译器，将其和 powerDiff.cpp, main.cpp 等文件一起编译链接成最终的 power_diff_test 可执行程序。

5.1.5.2 框架集成

为了将前述 PowerDifference 算子集成至 TensorFlow 框架中，需要完成 PluginOp 接口封装、DLP 算子集成和算子测试等步骤。

1. PluginOp 接口封装

如前所述，CNML 通过 PluginOp 相关接口提供了用户自定义算子和高性能库已有算子协同工作机制。因此，在完成 PowerDifference 算子的开发后，可以利用 CNML PluginOp 相关接口封装出方便用户使用的 CNPlugin 接口（包括 PluginOp 的创建、计算和销毁等接口），使用户自定义算子和高性能库已有算子有一致的编程模式和接口。

```

1 // filename: powerDiff.cpp
2 #include <stdlib.h>
3 #include "cnrt.h"
4 #include "cnrt_data.h"
5 #include "stdio.h"
6 #ifdef __cplusplus
7 extern "C" {
8 #endif
9 void PowerDifferenceKernel(half* input1, half* input2, int32_t pow, half* output,
10   int32_t len);
11 }
12 #endif
13 void PowerDifferenceKernel(half* input1, half* input2, int32_t pow, half* output,
14   int32_t len);
15
16 int MLUPowerDifferenceOp(float* input1, float* input2, int pow, float*output, int dims_a)
17 {
18   //some definition
19   //prepare data
20   if (CNRT_RET_SUCCESS != cnrtMalloc((void**)&mlu_input1, dims_a * sizeof(half))) {
21     printf("cnrtMalloc Failed!\n");
22     exit(-1);
23   }
24   ...
25   //kernel parameters
26   cnrtKernelParamsBuffer_t params;
27   cnrtGetKernelParamsBuffer(&params);
28   cnrtKernelParamsBufferAddParam(params, &mlu_input1, sizeof(half));
29   ...
30   //启动 Kernel
31   cnrtInvokeKernel_V2();
32   //将计算结果拷回 Host
33   cnrtMemcpy(_____, CNRT_MEM_TRANS_DIR_DEV2Host);
34   cnrtConvertHalfToFloatArray(_____);
35   // free data
36   cnrtDestroy();
37   ...
38   return 0;
39 }
```

图 5.11 调用运行时库函数的程序示例代码

图 5.14给出了 PluginOp 接口封装的部分示例代码，主要包括算子构建接口 Create、单算子运行接口 Compute 函数的具体实现。函数定义在 plugin_power_difference_op.cc 中，声明在 cnplugin.h 中。

- 算子构建接口 Create 函数：通过调用 cnmlCreatePluginOp 传递 BCL 算子函数指针、输入和输出变量指针完成算子创建。创建成功后可以得到 cnmlBaseOp_t 类型的指针。算子的相关参数需要使用 cnrtKernelParamsBuffer_t 的相关数据结构和接口创建。

- 单算子运行接口 Compute 函数：通过调用 cnmlComputePluginOpForward 利用前面创建的 cnmlBaseOp_t 的指针和输入输出变量指针完成上述计算过程。注意单独的 Compute 函数主要是在非融合模式下使用。

由于本算子的功能本身比较简单，所以参数（例如 power 和 len）采用了在 Create 时直接传递的方式。如果参数比较复杂则建议使用 OpParam 机制，将参数打包定义结构体来完

```

1 // filename: main.cpp
2 #include <math.h>
3 #include <time.h>
4 #include "stdio.h"
5 #include <stdlib.h>
6 #include <sys/time.h>
7
8 #define DATA_COUNT 32768
9 #define POW_COUNT // some num
10 int MLUPowerDifferenceOp( float* input1 , float* input2 , int pow , float*output , int dims_a)
11     ;
12
13 int main() {
14     // define
15     ...
16     FILE* f_input_x = fopen("./data/in_x.txt", "r");
17     FILE* f_input_y = fopen("./data/in_y.txt", "r");
18     FILE* f_output_data = fopen("./data/out.txt", "r");
19     struct timeval tpstart , tpend;
20     // load data
21     ...
22     gettimeofday(&tpstart , NULL);
23     MLUPowerDifferenceOp(input_x ,input_y ,POW_COUNT, output_data ,DATA_COUNT);
24     gettimeofday(&tpend , NULL);
25     time_use = 1000000 * (tpend.tv_sec - tpstart.tv_sec)+ tpend.tv_usec - tpstart.tv_usec;
26     for(int i = 0; i < DATA_COUNT; ++i)
27     {
28         err +=fabs(output_data_cpu[i] - output_data[i]);
29         cpu_sum +=fabs(output_data_cpu[i]);
30     }
31     printf("err rate = %0.4f%%\n" , err*100.0/cpu_sum);
32     return 0;
33 }
```

图 5.12 Main 程序示例代码

```

1 cncc -c --bang-mlu-arch=MLU200 plugin_power_difference_kernel.mlu -o powerdiffkernel.o
2 g++ -c main.cpp
3 g++ -c powerDiff.cpp -I/usr/local/neuware/include
4 g++ powerdiffkernel.o main.o powerDiff.o -o power_diff_test -L /usr/local/neuware/lib64
    -lcrt
```

图 5.13 测试程序的编译

成参数传递。

2. DLP 算子集成

为了使 DLP 硬件往 TensorFlow 框架中的集成更加模块化，我们对高性能库 CNML 算子进行了多个层次的封装，自顶向下包含以下几个层次：

- 最终运行的算子类 MLUOpKernel：继承 TensorFlow 中的 OpKernel 类，作为与 TensorFlow 算子层的接口；
- 封装 MLUStream 成员函数：与 MLUOpKernel 类接口关联，负责 MLU 算子的实例化并与运行时队列结合；
- 定义 MLUOps：负责 TensorFlow 算子的 DLP 实现，可以是单算子也可以是内存拼接的算子。完成对底层算子的调用后实现完整 TensorFlow 算子的功能供 MLUStream 部分调

```
1 // filename: plugin_power_difference_op.cc
2
3 // cnmlCreatePluginPowerDifferenceOp
4 cnmlStatus_t cnmlCreatePluginPowerDifferenceOp(
5     cnmlBaseOp_t *op,
6     cnmlTensor_t* input_tensors ,
7     int power,
8     cnmlTensor_t* output_tensors ,
9     int len
10) {
11     void** InterfacePtr;
12     InterfacePtr = reinterpret_cast<void*>(&PowerDifferenceKernel);
13     // Passing param
14     cnrtKernelParamsBuffer_t params;
15     cnrtGetKernelParamsBuffer(&params);
16     cnrtKernelParamsBufferMarkInput(params);    // input 0
17     cnrtKernelParamsBufferMarkInput(params);    // input 1
18     cnrtKernelParamsBufferAddParam(params, &power, sizeof(int));
19     cnrtKernelParamsBufferMarkOutput(params);   // output 0
20     cnrtKernelParamsBufferAddParam(params, &len, sizeof(int));
21     cnmlCreatePluginOp(op,
22                         "PowerDifference",
23                         InterfacePtr,
24                         params,
25                         input_tensors ,
26                         2,
27                         output_tensors ,
28                         1,
29                         nullptr ,
30                         0);
31     cnrtDestroyKernelParamsBuffer(params);
32     return CNML_STATUS_SUCCESS;
33 }
34 // cnmlComputePluginPowerDifferenceOpForward
35 cnmlStatus_t cnmlComputePluginPowerDifferenceOpForward(
36     cnmlBaseOp_t op,
37     void **inputs ,
38     void **outputs ,
39     cnrtQueue_t queue
40) {
41     cnmlComputePluginOpForward_V4(op,
42                                   nullptr ,
43                                   inputs ,
44                                   2,
45                                   nullptr ,
46                                   outputs ,
47                                   1,
48                                   queue ,
49                                   nullptr );
50     return CNML_STATUS_SUCCESS;
51 }
```

图 5.14 PluginOp 接口封装的示例代码

用；

- 对 CNPlugin 封装的 MLULib：对 CNML 和 CNRT 接口的直接封装供 MLUOps 调用，只包含极少的 TensorFlow 数据结构。

上述四个层次自顶向下连接了 TensorFlow 内部的 OpKernel 和 DLP 所提供的高性能库及运行时库，因此在 TensorFlow 中集成 DLP 算子涉及上面各层次。集成的整体流程如图 5.15 所示，主要包括：算子注册、定义 MLULib 层接口、定义 MLUOps 层接口、定义 MLUStream 层接口以及定义 MLUOpKernel 层接口并注册。

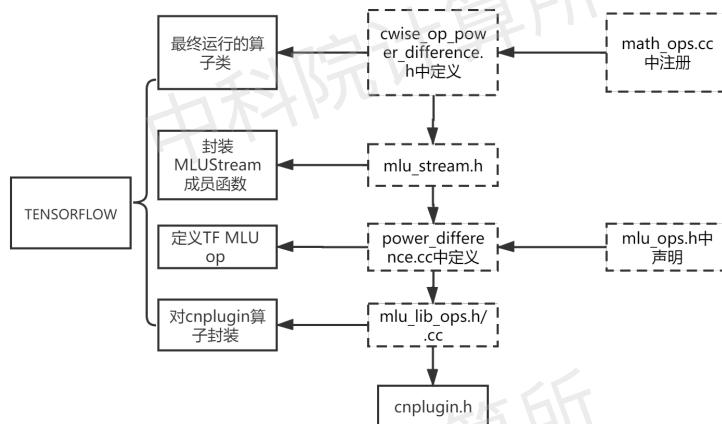


图 5.15 TensorFlow 集成流程图

• 算子注册

参考第4.4节中注册的CPU算子，在 tensorflow/core/kernels/cwise_op_power_difference.cc 文件中添加如图 5.16 所示的 DLP 算子 Kernel 的注册信息 (REGISTER_KERNEL_BUILDER)。此外，DLP 算子会与 CPU 算子共享在 tensorflow/core/ops/math_ops.cc 中的算子注册方法 (图 4.26 所示的 REGISTER_OP)，这样用户可以使用相同的 Python API (power_difference) 调用自定义算子，在编程上无需感知底层硬件的差异。

```

1 // filename: tensorflow/core/kernels/cwise_op_power_difference.cc
2 #define REGISTER_MLU(T) \
3     REGISTER_KERNEL_BUILDER( \
4         Name("PowerDifference") \
5             .Device(Device_MLU) \
6             .TypeConstraint<T>("T"), \
7             MLUPowerDifferenceOp<T>; \
8 TF_CALL_MLU_FLOAT_TYPES(REGISTER_MLU);

```

图 5.16 TensorFlow 框架中集成 DLP 算子 (1)：算子注册

• 定义 MLULib 层接口

定义 MLULib 层接口主要是将前述已通过 PluginOp 接口封装好的接口如 cnmlCreatePluginInPowerDifferenceOp 和 cnmlComputePluginPowerDifferenceOpForward 与 TensorFlow 中的 MLULib 层接口进行绑定，实现 MLULib 层的 CreatePowerDifferenceOp 和 ComputePowerDifferenceOp。该部分代码位于 tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.h。

tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.cc 和 tensorflow/stream_executor/mlu/mlu_api/mlu.h 中。具体如图 5.17 所示。

```

1  ##tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.h
2  tensorflow::Status CreatePowerDifferenceOp(MLUBaseOp** op, MLUTensor* input1, MLUTensor*
   input2, int input3, MLUTensor* output, int len);
3  tensorflow::Status ComputePowerDifferenceOp(MLUBaseOp* op, MLUCnrtQueue* queue, void*
   input1, void* input2, void* output);
4
5  ##tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.cc
6  tensorflow::Status CreatePowerDifferenceOp(MLUBaseOp** op, MLUTensor* input1, MLUTensor*
   input2, int input3, MLUTensor* output, int len) {
7    MLUTensor* inputs_ptr[2] = {input1, input2};
8    MLUTensor* outputs_ptr[1] = {output};
9    CNML_RETURN_STATUS(cnmlCreatePluginPowerDifferenceOp(op, inputs_ptr, input3,
10                      outputs_ptr, len));
11}
12 tensorflow::Status ComputePowerDifferenceOp(MLUBaseOp* op, MLUCnrtQueue* queue, void*
   input1, void* input2, void* output) {
13  void* inputs_ptr[2] = {input1, input2};
14  void* outputs_ptr[1] = {output};
15  CNML_RETURN_STATUS(cnmlComputePluginPowerDifferenceOpForward(op, inputs_ptr,
16                      outputs_ptr, queue));
17}
18 ###tensorflow/stream_executor/mlu/mlu_api/ops/mlu_ops.h 中添加声明
19 DECLARE_OP_CLASS(MLUPowerDifference);

```

图 5.17 TensorFlow 框架中集成 DLP 算子 (2): 定义 MLULib 层接口

• 定义 MLUOp 层接口

定义 MLUOp 层接口主要是在 MLUOp 层实现算子类的 Create 和 Compute 等方法。该部分代码位于 tensorflow/stream_executor/mlu/mlu_api/ops/power_difference.cc 文件中。其中 CreateMLUOp 和 Compute 等方法将调用前面在 MLULib 层实现好的 CreatePowerDifferenceOp 和 ComputePowerDifferenceOp 等方法。具体代码如图 5.18 所示。

• 定义 MLUStream 层接口

定义 MLUStream 层接口主要是在 MLUStream 层(tensorflow/stream_executor/mlu/mlu_stream.h)添加算子类声明。其与 MLUOpKernel 类接口关联，负责 MLU 算子的实例化。在运行时这层代码会自动将算子与运行时队列进行绑定并下发执行。具体代码如图 5.19 所示。

• 定义 MLUOpKernel 层接口

定义 MLUOpKernel 层接口主要是在 MLUOpKernel 层定义 MLUPowerDifferenceOp，在其中通过 stream 机制调用 MLUStream 层具体的 PowerDifference 函数。该部分代码位于 tensorflow/core/kernels/cwise_op_power_difference_mlu.h 具体代码如图 5.20 所示。

3. 算子测试

在新增自定义的 PowerDifference 算子与 TensorFlow 框架的集成完后，用户需要使用 Bazel 重新编译 TensorFlow，然后即可使用 Python 侧的 API 对新集成的算子功能进行测试。由于对用户的 API 是一致的，用户在测试时需要通过环境变量来配置该算子的实现是调用 CPU 还是 DLP 版本。该部分代码位于 power_difference_test_bcl.py。完整的单算子 Python 测试代码如图 5.21 所示。

```

1 // filename: tensorflow/stream_executor/mlu/mlu_api/ops/power_difference.cc
2 Status MLUPowerDifference::CreateMLUOp(std::vector<MLUTensor *> &inputs, std::vector<
   MLUTensor *> &outputs, void *param) {
3   TF_PARAMS_CHECK(inputs.size() > 1, "Missing input");
4   TF_PARAMS_CHECK(outputs.size() > 0, "Missing output");
5   MLUBaseOp *power_difference_op_ptr = nullptr;
6   MLUTensor *input1 = inputs.at(0);
7   MLUTensor *input2 = inputs.at(1);
8   int power_c = *((int *)param);
9   MLUTensor *output = outputs.at(0);
10 ...
11   TF_STATUS_CHECK(lib::CreatePowerDifferenceOp(____));
12 ...
13 }
14 Status MLUPowerDifference::Compute(const std::vector<void *> &inputs, const std::vector<
   void *> &outputs, cnrtQueue_t queue) {
15 ...
16   TF_STATUS_CHECK(lib::ComputePowerDifferenceOp(____));
17 ...
18 }
```

图 5.18 TensorFlow 框架中集成 DLP 算子 (3): 定义 MLUOp 层接口

```

1 // filename: tensorflow/stream_executor/mlu/mlu_stream.h
2 Status PowerDifference(OpKernelContext* ctx,
3   Tensor* input1, Tensor* input2, Tensor* output, int input3) {
4   return CommonOpImpl<ops::MLUPowerDifference>(ctx,
5     {input1, input2}, {output}, static_cast<void*>(&input3));
6 }
```

图 5.19 TensorFlow 框架中集成 DLP 算子 (4): 定义 MLUStream 层接口

```

1 // filename: tensorflow/core/kernels/cwise_op_power_difference_mlu.h
2 class MLUPowerDifferenceOp : public MLUOpKernel {
3 public:
4   explicit MLUPowerDifferenceOp(OpKernelConstruction* ctx) :
5     MLUOpKernel(ctx) {}
6   void ComputeOnMLU(OpKernelContext* ctx) override {
7     // 输入数据处理与条件判断
8     -----
9     // Stream 调用 PowerDifference 接口
10    OP_REQUIRES_OK(ctx, stream->PowerDifference(____));
```

图 5.20 DLP 算子集成 (5): 定义 MLUOpKernel 层接口

5.1.5.3 在线推理

针对完整的 pb 模型推理，在框架层集成了 DLP 算子后，在创建 TensorFlow 的执行图时，会自动将这些算子分配到 DLP 上计算，无需使用者显式指定。具体而言，只需在第 4.4 节的实验基础上，使用新编译的 TensorFlow 重复执行一次即可。可以看到，新集成了 DLP 上的 PowerDifference 算子后，整个 pb 模型可以完整地跑在 DLP 上，且性能相较于纯 CPU 版本（第 4.2 节）和部分 CNML 版本（第 4.4 节）都有显著的提升。

```

1 #power_difference_test_bcl.py
2 import numpy as np
3 import os
4 import time
5 #使用以下环境变量控制单算子的执行方式
6 os.environ['MLU_VISIBLE_DEVICES']="0"
7 os.environ['TF_CPP_MIN_MLU_LOG_LEVEL']="1"
8 import tensorflow as tf
9 np.set_printoptions(suppress=True)
10
11 def power_difference_op(input_x, input_y, input_pow):
12     with tf.Session() as sess:
13         x = tf.placeholder(tf.float32, name='x')
14         y = tf.placeholder(tf.float32, name='y')
15         pow_ = tf.placeholder(tf.float32, name='pow')
16         z = tf.power_difference(x,y,pow_)
17         return sess.run(z, feed_dict = {x: input_x, y: input_y, pow_: input_pow})
18
19 def main():
20     start = time.time()
21     input_x = np.loadtxt("./data/in_x.txt", delimiter=',')
22     input_y = np.loadtxt("./data/in_y.txt")
23     input_pow = np.loadtxt("./data/in_z.txt")
24     output = np.loadtxt("./data/out.txt")
25     end = time.time()
26     print("load data cost " + str((end-start)*1000) + "ms")
27     start = time.time()
28     res = power_difference_op(input_x, input_y, input_pow)
29     end = time.time()
30     print("comput op cost " + str((end-start)*1000) + "ms")
31     err = sum(abs(res - output))/sum(output)
32     print("err rate= " + str(err*100))
33
34 if __name__ == '__main__':
35     main()

```

图 5.21 采用 Python API 对集成的单算子进行测试

5.1.5.4 离线推理

通过前一小节的在线推理，可以得到不分段实时风格迁移的离线模型。在实际场景中，为了尽可能提高部署的效率，通常会选择离线部署的方式。离线与在线的区别在于其脱离了 TensorFlow 编程框架和高性能库 CNML，仅与运行时库 CNRT 相关，减少了不必要的开销，提升了执行效率。

在编写离线推理工程时，DLP 目前仅支持 C++ 语言。与在线推理相似，离线推理主要包含：输入数据前处理、离线推理及后处理。下面详细介绍具体的实现代码。

1. 主函数

主函数主要用于串联整体流程，该部分代码位于 `src/style_transfer.cpp`。具体如图 5.22 所示。

2. 数据前处理

常见的数据前处理包括减均值、除方差、图像大小 Resize、图像数据类型转换（例如 Float 和 INT 转换）、RGB 转 BGR 转换等等。具体需要哪些预处理需要与原神经网络模型对齐。以 Resize 操作为例，可以调用 OpenCV 中的 Resize 函数 `cv::resize(sample, sample_resized,`

```

1 // filename: src/style_transfer.cpp
2 #include "style_transfer.h"
3 #include <math.h>
4 #include <time.h>
5 #include "stdio.h"
6 #include <stdlib.h>
7 #include <sys/time.h>
8
9 int main(int argc, char** argv){
10     // parse args
11     std::string file_list = "/path/to/images/" + std::string(argv[1]) + ".jpg";
12     std::string offline_model = "/path/to/models/offline_models/" + std::string(argv[2])
13         + ".cambricon";
14
15     // creat data
16     DataTransfer* DataT = (DataTransfer*) new DataTransfer();
17     DataT->image_name = argv[1];
18     DataT->model_name = argv[2];
19     // process image
20     DataProvider *image = new DataProvider(file_list);
21     image->run(DataT);
22
23     // running inference
24     Inference *infer = new Inference(offline_model);
25     infer->run(DataT);
26
27     // postprocess image
28     PostProcessor *post_process = new PostProcessor();
29     post_process->run(DataT);
30
31     delete DataT;
32     DataT = NULL;
33     delete image;
34     image = NULL;
35     delete infer;
36     infer = NULL;
37     delete post_process;
38     post_process = NULL;
39 }

```

图 5.22 DLP 离线部署主函数

cv::Size(256,256)); 该函数参数分别对应输入、输出和 Resize 的目标大小等。该部分代码位于 src/data_provider.cpp 中。

3. 离线推理

离线推理部分主要是使用 CNRT API 运行离线模型。其主要流程包括以下步骤：

第一步将磁盘上的离线模型文件载入并抽取出 CNRT Function。一个离线模型文件中可以存储多个 Function，但是多数情况下离线模型文件中只有一个 Function，这取决于离线模型生成时框架层的设置。本实验中由于所有算子都可以在 DLP 上运行，经过 CNML 算子间融合处理之后只有一个 Function。

第二步要准备 Host 与 Device 的输入输出内存空间和数据。由于 DLP 的异构计算特征，需要先在 Host 端准备好数据后再将其拷贝到 Device 端，所以在此之前也要先分别在 Device 端和 Host 端分配相应内存空间。其中需要注意的是数据类型（例如 INT 或 Float）和存储格式（例如 NCHW 或 NHWC）在 Host 端和 Device 端之间可能会不同，所以在做数据拷贝

```

1 // filename: data_provider.cpp
2 #include "data_provider.h"
3
4 namespace StyleTransfer{
5 DataProvider :: DataProvider(std::string file_list_){
6     ...
7     set_mean();
8 }
9 void DataProvider :: set_mean(){
10    float mean_value[3] = {
11        0.0,
12        0.0,
13        0.0,
14    };
15    cv::Mat mean(256, 256, CV_32FC3, cv::Scalar(mean_value[0], mean_value[1], mean_value
16 [2]));
17    mean_ = mean;
18 }
19 bool DataProvider :: get_image_file(){
20     image_list.push_back(file_list);
21     return true;
22 }
23 cv::Mat DataProvider :: convert_color_space(std::string file_path){
24     cv::Mat sample;
25     cv::Mat img = cv::imread(file_path, -1);
26     ...
27     return sample;
28 }
29 cv::Mat DataProvider :: resize_image(const cv::Mat& source){
30     cv::Mat sample_resized;
31     cv::Mat sample;
32     ...
33     return sample_resized;
34 }
35 cv::Mat DataProvider :: convert_float(cv::Mat img){
36     cv::Mat float_img;
37     ...
38     return float_img;
39 }
40 cv::Mat DataProvider :: subtract_mean(cv::Mat float_image){...}
41 void DataProvider :: split_image(DataTransfer* DataT){...}
42 DataProvider :: ~DataProvider(){}
43 void DataProvider :: run(DataTransfer* DataT){
44     for(int i = 0; i < batch_size; i++){
45         get_image_file();
46         std::string img_path= image_list[i];
47         cv::Mat img_colored = convert_color_space(img_path);
48         cv::Mat img_resized = resize_image(img_colored);
49         cv::Mat img_floated = convert_float(img_resized);
50         DataT->image_processed.push_back(img_floated);
51     }
52     split_image(DataT);
53 }
54 }
```

图 5.23 DLP 离线部署数据前处理

前要先完成相应的转换。

第三步主要和 DLP 设备本身相关。包括设置运行时上下文、绑定设备、将计算任务下

发到队列等。

第四步将计算结果拷回 Host 端并完成相关的数据转换。

最后一步将上面申请的所有内存和资源释放。

上述代码位于 src/inference.cpp 中。

```
1 // filename: inference.cpp
2 #include "inference.h"
3 #include "cnrt.h"
4 ...
5 namespace StyleTransfer{
6 Inference :: Inference(std::string offline_model){
7     offline_model_ = offline_model;
8 }
9 void Inference :: run(DataTransfer* DataT){
10    // load model
11    // load extract function
12    // prepare data on cpu
13    // allocate I/O data memory on DLP
14    // prepare input buffer
15    // prepare output buffer
16    // setup runtime ctx
17    // bind Device
18    // compute offline
19    // free memory spac
20 }
21 } // namespace StyleTransfer
```

图 5.24 DLP 离线部署推理

4. 后处理

这部分主要完成将计算结果保存成图片，具体代码位于 src/post_processor.cpp 中。

5. 编译运行

这里借助 CMake 工具完成对整个项目的编译管理，具体代码在 CMakeList.txt 中。

5.1.6 实验评估

本次实验中主要考虑基于智能编程语言的算子实现与验证、与框架的集成以及完整的模型推理。模型推理的性能和精度应同时作为主要的参考指标。因此，本实验的评估标准设定如下：

- 60 分标准：完成 PowerDifference 算子实现以及基于 CNRT 的测试，在测试数据中精度误差在 1% 以内，延时在 100ms 以内；
- 80 分标准：在 60 分基础上，完成 BCL 算子与 TensorFlow 框架的集成，使用 Python 在 Device 端测试大规模数据时，精度误差在 10% 以内，平均延时在 150ms 以内。
- 90 分标准：在 80 分基础上，使用 DLP 推理完整 pb 模型时，输出精度正常的风格迁移图片，输出正确的离线模型。
- 100 分标准：在 90 分基础上，完成离线推理程序的编写，执行离线推理时风格迁移图片精度正常。

```

1 // filename: post_processor.cpp
2 #include "post_processor.h"
3
4 namespace StyleTransfer{
5
6 PostProcessor :: PostProcessor(){
7     std::cout << "PostProcessor constructor" << std::endl;
8 }
9
10 void PostProcessor :: save_image(DataTransfer* DataT){
11
12     std::vector<cv::Mat> mRGB(3);
13     for(int i = 0; i < 3; i++){
14         cv::Mat img(256, 256, CV_32FC1, DataT->output_data + 256 * 256 * i);
15         mRGB[i] = img;
16     }
17     cv::Mat im(256, 256, CV_8UC3);
18     cv::merge(mRGB, im);
19
20     std::string file_name = DataT->image_name + std::string("_") + DataT->model_name + ".jpg";
21     cv::imwrite(file_name, im);
22     std::cout << "style transfer result file: " << file_name << std::endl;
23 }
24
25 PostProcessor :: ~PostProcessor(){
26     std::cout << "PostProcessor destructor" << std::endl;
27 }
28
29 void PostProcessor :: run(DataTransfer* DataT){
30     save_image(DataT);
31 }
32
33 } // namespace StyleTransfer

```

图 5.25 DLP 离线部署后处理

5.1.7 实验思考

1. PowerDifference 算子实现本身性能提升有哪些方法?
2. 融合方式为何可以带来性能的提升?
3. 离线方式为何可以带来性能的提升?
4. 如何更好地利用 DLP 的多核架构来提升性能?

```

1 // filename: CMakeLists.txt
2 cmake_minimum_required(VERSION 2.8)
3 project(style_transfer)
4
5 set(CMAKE_BUILD_TYPE "Debug")
6 set(CMAKE_CXX_FLAGS_DEBUG "-std=c++11 -g -Wall ${CMAKE_CXX_FLAGS_DEBUG}")
7 set(CMAKE_EXE_LINKER_FLAGS "-lpthread -fPIC ${CMAKE_EXE_LINKER_FLAGS}")
8
9 find_package(OpenCV REQUIRED COMPONENTS core imgproc highgui)
10 include_directories(${OPENCV_INCLUDE_DIR})
11 include_directories(${ENV{NEUWARE}/include})
12 include_directories(${CMAKE_CURRENT_SOURCE_DIR}/include)
13
14 #link_directories(${CMAKE_CURRENT_SOURCE_DIR}/lib)
15 link_directories(${ENV[X86_LIB_PATH]})
16 link_directories(${ENV{NEUWARE}/lib64})
17 link_libraries("libcrt.so")
18 set(EXECUTABLE_OUTPUT_PATH ${CMAKE_CURRENT_SOURCE_DIR}/bin)
19
20 add_executable(style_transfer src/style_transfer.cpp
21           src/data_provider.cpp
22           src/inference.cpp
23           src/post_processor.cpp)
24
25 target_link_libraries(style_transfer ${OpenCV_LIBS})

```

图 5.26 DLP 离线部署编译-CMakeLists.txt

5.2 智能编程语言性能优化实验

5.2.1 实验目的

掌握使用智能编程语言优化算法性能的原理，掌握智能编程语言的调试和调优方法，能够使用智能编程语言在 DLP 上加速矩阵乘的计算。

实验工作量：约 70 行代码，6 小时。

5.2.2 背景介绍

5.2.2.1 BCL 编程模型

BCL 编程模型的主要内涵在智能计算系统教材中已经全面阐述过了，在这里简单回顾一下和本实验相关的概念。

如图5.27所示，在本实验中同学们需要手动完成数据在 Global Memory (GDRAM)，SRAM，NRAM，WRAM 之间的调度。

在 DLP 中，每个 cluster 中包含 4 个计算核，1 个 SRAM。需要注意的是，在每个 Cluster 中和 SRAM 相配合的还有 1 个 Memory Core 的部件，专门用于管理片上总线和 SRAM。与图中普通的计算 Core 相比，Memory Core 本身不承担计算任务，但是可以独立运行，完成数据从 GDRAM 到 SRAM 的拷贝。所以，在实际运行时 Memory Core 可以承担一部分数据搬运的任务，减轻计算 Core 的负担。在具体编程时，只要调用 BCL 的 memcpy 函数，如果拷贝方向是 GDRAM2SRAM，那么在编译器编译代码时就会自动将相关指令放到 Memory core 上去执行。

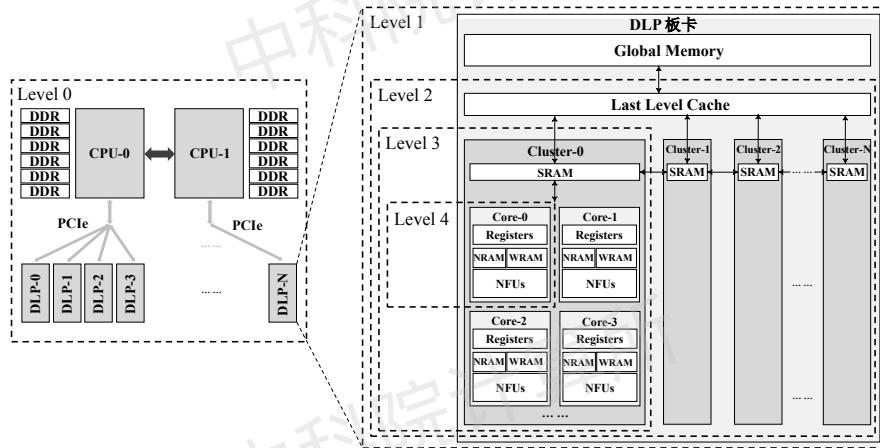


图 5.27 BCL 编程模型

5.2.2.2 DLP 并行编程

在 BCL 中使用一系列的并行内嵌变量来帮助使用者充分发挥 DLP 的并行特征。

Core 变量：

coreDim 表示一个 Cluster 包含的 Core 个数，MLU270 上等于 4

coreId 表示每个 Core 在 Cluster 内的逻辑 ID，MLU270 上的取值范围为 [0-3]

Cluster 变量：

clusterDim 表示启动 Kernel 时指定的任务调度型表示的 Cluster 个数，例如 UNION4 时等于 4

clusterId 表示 clusterDim 内某个 Cluster 的逻辑 ID，例如 UNION4 时其取值范围是 [0-3]

Task 变量：

taskDimX/taskDimY/taskDimZ 表示 1 个 Task 在 [XYZ] 方向上的任务规模，其值等于 Host 端所指定的任务规模

taskDim 表示任务线性化后的任务规模信息，线性化公式如下：

$$\text{taskDim} = \text{taskDimX} * \text{taskDimY} * \text{taskDimZ}$$

taskIdX taskIdY taskIdZ 表示程序运行时所分配的逻辑规模在 [XYZ] 方向上的任务 ID

taskId 表示程序运行时所分配的任务 ID，其值为对逻辑规模降维后的任务 ID：

$$\text{taskId} = \text{taskIdZ} * \text{taskDimY} * \text{taskDimX} + \text{taskIdY} * \text{taskDimX} + \text{taskIdX}$$

如图5.2是一个实际的内嵌变量取值示例。当程序调用 8 个计算核（UNION2）的时候，每个核上的并行变量就会得到如图所示的取值。这里 taskDimX,Y,Z 设为 {8,1,1}。

表 5.2 DLP 并行内嵌变量示例

taskId	taskIdX	taskIdY	taskIdZ	clusterDim	coreDim	coreId	clusterID	taskDimX	taskDimY	taskDimZ	taskDim
0	0	0	0	2	4	0	0	8	1	1	8
1	1	0	0	2	4	1	0	8	1	1	8
2	2	0	0	2	4	2	0	8	1	1	8
3	3	0	0	2	4	3	0	8	1	1	8
4	4	0	0	2	4	0	1	8	1	1	8
5	5	0	0	2	4	1	1	8	1	1	8
6	6	0	0	2	4	2	1	8	1	1	8
7	7	0	0	2	4	3	1	8	1	1	8

5.2.2.3 Notifier 机制

在本实验中不再要求框架集成等系统开发内容，主要思路集中在智能编程语言的使用和优化。为了详细地统计程序在 DLP 上的运行时间，在此我们介绍 Notifier 机制。

Notifier 可以看作一种特殊类型的任务，它可以像 Kernel 任务一样放入 Queue 中执行。无论是 Notifier 还是 Kernel，内部队列始终遵循 FIFO 调度原则。与 Kernel 任务相比，Notifier 任务不需要执行实际的硬件操作，只占用很少的执行时间（几乎可以忽略不计）。可以使用 Notifier 任务来统计 Kernel 计算任务的硬件执行时间。图 5.28 展示了对 ROI Pooling Kernel 的执行时间进行统计的示例。

```

1 cnrtNotifier_t notifier_start, notifier_end;
2 cnrtCreateNotifier(&notifier_start);
3 cnrtCreateNotifier(&notifier_end);
4 cnrtPlaceNotifier(notifier_start, pQueue);
5 ret = cnrtInvokeKernel_V3(reinterpret_cast<void *>(&ROI Pooling Kernel), init_param, dim,
   params, c, pQueue, NULL);
6 cnrtPlaceNotifier(notifier_end, pQueue);
7 ret = cnrtSyncQueue(pQueue);
8 cnrtNotifierElapsedTime(notifier_start, notifier_end, &timeTotal);
9 printf("Hardware Total Time: %.3f ms\n", timeTotal / 1000.0);

```

图 5.28 Notifier 机制代码示例

5.2.3 实验环境

本节实验所涉及的硬件平台和软件环境如下：

- 硬件平台：硬件平台基于前述的 DLP 云平台环境。
- 软件环境：所涉及的 DLP 软件开发模块包括编程语言及编译器，高性能库 CNML 和运行时库 CNRT 仅作参考学习。

5.2.4 实验内容

本节实验内容主要为矩阵乘法的性能优化。需要完成如图 5.29 的矩阵乘法 Host 端和 Device 端的代码并正确编译运行。本实验中的 Device 端代码只接受特定规模的输入。

在计算结果正确的基础上，需要进一步使用多种手段完成性能优化，提升计算效率。主要的优化手段包括：

- 充分利用 DLP 的片上存储架构，包括 NRAM 和 WRAM 等特性实现高效数据调度。
- 在片上数据调度的基础上使用 `_bang_conv` 张量操作函数实现高效矩阵乘运算。
- 在充分利用单核计算架构（架构特点包括了 NRAM、WRAM 和张量计算指令等）的基础上，借助 DLP 本身的多核架构实现任务并行。
- 在朴素的多核并行的基础上，更进一步使用 Cluster 架构，使用片上 SRAM 达到减轻核间访存带宽竞争的目的。
- 充分使用片上 Memory Core 架构实现访存和计算流水化作业。

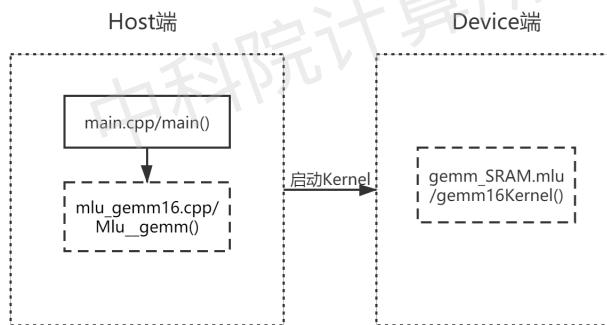


图 5.29 实验内容

本节实验涉及到的文件比较简单，图5.29所示，主要为 Host 端的 cpp 文件和 Device 端的 mlu 文件。

5.2.5 实验步骤

结合前述优化手段，本节实验的具体步骤包括：Host 端程序实现、标量操作实现、张量操作实现、多核并行实现、SRAM 使用、访存和计算流水等。

5.2.5.1 Host 端程序

由于智能编程语言采用异构编程，一个完整的程序包括 Host 端和 Device 端。Host 端和 Device 端分别进行编程和编译，最后链接成一个可执行程序。Host 端使用 C/C++ 语言进行编写，调用 CNRT 接口执行控制部分和串行任务；Device 端使用 BCL 特定的语法规则执行计算部分和并行任务。用户可以在 Host 端输入数据，做一定处理后，通过一个 Kernel 启动函数将相应输入数据传给 Device 端，Device 端进行计算后，再将计算结果拷回 Host 端。

在整个矩阵乘程序执行过程中，用户先输入参数 m, k, n 代表要计算的左右矩阵分别为 $m * k$ 和 $k * n$ 大小，随后 Host 端对这两个矩阵进行随机赋值，将输入矩阵及大小相应的参数传入 Device 端进行矩阵运算，然后将运算结果传回 Host 端，在 Host 端打印矩阵乘的硬件执行时间。Host 端代码位于 `main.cpp` 和 `mlu_gemm16.cpp`。其关键代码如矩阵乘 Host 端代码示例所示：

```

1 // filename: mlu_gemm16.cpp
2 #include <float.h>
3

```

```
4 #include <math.h>
5 #include <memory.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <sys/time.h>
9 #include <vector>
10 #include "cnrt.h"
11 #include "gemm16Kernel.h"
12
13 #define PAD_UP(x, m) ((x + m - 1) / m * m)
14 #define MP_SELECT 16
15 #define MP1 ((MP_SELECT & 1))
16 #define MP4 ((MP_SELECT & 4))
17 #define MP8 ((MP_SELECT & 8))
18 #define MP16 ((MP_SELECT & 16))
19 #define MP32 ((MP_SELECT & 32))
20 int Mlu_gemm(int8_t *A, int8_t *B, float *C, int32_t M, int32_t N, int32_t K,
21     int16_t pos1, int16_t pos2, float scale1, float scale2, float &return_time) {
22     struct timeval start;
23     struct timeval end;
24     float time_use;
25     int N_align = N;
26     cnrtRet_t ret;
27     gettimeofday(&start, NULL);
28
29     cnrtQueue_t pQueue;
30     CNRT_CHECK(cnrtCreateQueue(&pQueue));
31
32     cnrtDim3_t dim;
33     cnrtFunctionType_t func_type = CNRT_FUNC_TYPE_BLOCK; // CNRT_FUNC_TYPE_BLOCK=1
34     dim.x = 1;
35     dim.y = 1;
36     dim.z = 1;
37
38     if (MP1) {
39         dim.x = 1;
40         func_type = CNRT_FUNC_TYPE_BLOCK;
41     } else if (MP4) {
42         dim.x = 4;
43         func_type = CNRT_FUNC_TYPE_UNION1;
44         // printf("UNION1!\n");
45     } else if (MP8) {
46         dim.x = 8;
47         func_type = CNRT_FUNC_TYPE_UNION2;
48     } else if (MP16) {
49         dim.x = 16;
50         func_type = CNRT_FUNC_TYPE_UNION4;
51         // printf("16\n");
52     } else if (MP32) {
53         dim.x = 32;
54         func_type = CNRT_FUNC_TYPE_UNION8;
55     } else {
56         printf("MP select is wrong! val = %d, use default setting ,mp=1\n",
57               MP_SELECT);
58         return -1;
59     }
60
61     gettimeofday(&end, NULL);
62     time_use =
63         ((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec)) /
64         1000.0;
65     // printf(" cnrt init time use %f ms\n", time_use);
```

```

66
67 float *h_f32b = (float *)malloc(K * sizeof(float));
68 half *h_c = (half *)malloc(M * N_align * sizeof(half));
69
70 half *d_c = NULL;
71 int8_t *d_a = NULL;
72 int8_t *d_w = NULL;
73 int16_t pos = pos1 + pos2;
74
75 gettimeofday(&start, NULL);
76
77 // 分配空间
78 CNRT_CHECK(cnrtMalloc((void **)&d_c, sizeof(half) * M * N_align));
79 CNRT_CHECK(cnrtMalloc((void **)&d_a, sizeof(int8_t) * M * K));
80 CNRT_CHECK(cnrtMalloc((void **)&d_w, sizeof(int8_t) * K * N_align));
81
82 // 将矩阵A和B的内容赋值给新分配的空间
83 CNRT_CHECK(cnrtMemcpy(d_a, A, sizeof(int8_t) * M * K, CNRT_MEM_TRANS_DIR_HOST2DEV));
84 CNRT_CHECK(cnrtMemcpy(d_w, B, sizeof(int8_t) * K * N_align, CNRT_MEM_TRANS_DIR_HOST2DEV));
85
86 gettimeofday(&end, NULL);
87 time_use =
88     (((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec)) /
89     1000.0;
90 // printf("malloc &copyin time use %f ms\n", time_use);
91
92 cnrtKernelParamsBuffer_t params;
93 CNRT_CHECK(cnrtGetKernelParamsBuffer(&params)); // Gets a parameter buffer for
94 // cnrtInvokeKernel_V2 or cnrtInvokeKernel_V3.
95 CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &d_c, sizeof(half))); // Adds a parameter
96 // to a specific parameter buffer.
97 CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &d_a, sizeof(int8_t *)));
98 CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &d_w, sizeof(int8_t *)));
99 CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &M, sizeof(uint32_t)));
100 CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &K, sizeof(uint32_t)));
101 CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &N_align, sizeof(uint32_t)));
102 CNRT_CHECK(cnrtKernelParamsBufferAddParam(params, &pos, sizeof(uint16_t)));
103
104 cnrtKernelInitParam_t init_param;
105 CNRT_CHECK(cnrtCreateKernelInitParam(&init_param));
106 CNRT_CHECK(cnrtInitKernelMemory((const void *)gemm16Kernel, init_param));
107
108 cnrtNotifier_t notifier_start; // A pointer which points to the struct describing notifier.
109 cnrtNotifier_t notifier_end;
110 CNRT_CHECK(cnrtCreateNotifier(&notifier_start));
111 CNRT_CHECK(cnrtCreateNotifier(&notifier_end));
112 float timeTotal = 0.0;
113
114 // printf("start invoke :\n");
115 gettimeofday(&start, NULL);
116
117 CNRT_CHECK(cnrtPlaceNotifier(notifier_start, pQueue)); // Places a notifier in specified queue
118 CNRT_CHECK(
119     cnrtInvokeKernel_V3((void *)&gemm16Kernel, init_param, dim, params, func_type, pQueue, NULL)
120 ); // Invokes a kernel written in Bang with given params on MLU
121 CNRT_CHECK(cnrtPlaceNotifier(notifier_end, pQueue)); // Places a notifier in specified queue
122
123 CNRT_CHECK(cnrtSyncQueue(pQueue)); // Function should be blocked until all precedent tasks in
124 // the queue are completed. 同步Queue
125 gettimeofday(&end, NULL);
126 time_use =
127     (((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec)) /

```

```

124     1000.0;
125 //  printf("invoke time use %f ms\n", time_use);
126 //  cnrtNotifierElapsedTime(notifier_start, notifier_end, &timeTotal);
127 //  get the duration time between notifer_start and notifer_end.
128 //  cnrtNotifierDuration(notifier_start, notifier_end, &timeTotal);    // Gets duration time of
   two makers
129 CNRT_CHECK(cnrtNotifierDuration(notifier_start, notifier_end, &timeTotal));
130 return_time = timeTotal / 1000.0;
131 // printf("hardware total Time: %.3f ms\n", return_time);
132 gettimeofday(&start, NULL);
133
134 CNRT_CHECK(cnrtMemcpy(h_c, d_c, sizeof(half) * M * N_align,
135                   CNRT_MEM_TRANS_DIR_DEV2HOST));
136 for (int j = 0; j < M; j++) {
137     for (int i = 0; i < N; i++) {
138         CNRT_CHECK(cnrtConvertHalfToFloat(&C[j * N + i], h_c[j * N_align + i]));
139         C[j * N + i] = C[j * N + i] / (scale1 * scale2);
140     }
141 }
142 gettimeofday(&end, NULL);
143 time_use =
144     ((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec)) /
145     1000.0;
146 // printf("copyout &convert time use %f ms\n", time_use);
147
148 // free
149 CNRT_CHECK(cnrtFree(d_c));
150 CNRT_CHECK(cnrtFree(d_a));
151 CNRT_CHECK(cnrtFree(d_w));
152
153 CNRT_CHECK(cnrtDestroyQueue(pQueue));
154 CNRT_CHECK(cnrtDestroyKernelParamsBuffer(params));
155 CNRT_CHECK(cnrtDestroyNotifier(&notifier_start));
156 CNRT_CHECK(cnrtDestroyNotifier(&notifier_end));
157 free(h_f32b);
158 free(h_c);
159 // free(h_w);
160 // free(h_w_reshape);
161 return 0;
162 }

```

Listing 1 矩阵乘 Host 端代码示例

在优化过程中，Host 端的代码基本不变，我们重点关注 Device 端代码的优化过程。

5.2.5.2 标量操作实现

该步骤直接在 NRAM 上使用循环和标量操作进行计算。每个 DLP 计算核都有自己的 NRAM，和 GDRAM 相比，虽然空间较小，但是具有更高的读写带宽和更低的访问延迟。因此，该步骤中将输入的左右矩阵全部从 GDRAM 拷入 NRAM 中，在 NRAM 中进行计算，然后再拷回 GDRAM。为了方便读者理解，在这个例子中我们假设输入的左右矩阵规模都为 256*256，以保证输入矩阵可以一次性拷入 NRAM。一旦输入矩阵规模超过 NRAM 的空间大小时，则需要对 NRAM 复用进行多次拷入和拷出。如图5.30的代码所示，该程序中无须对输入矩阵做任何处理，直接使用矩阵乘公式进行计算。由于并没有利用到 DLP 硬件架构的优势，计算时间较长。该代码命名为 gemm_GDRAM.mlu。

```

1 // filename: gemm_GDRAM.mlu
2 #include "mlu.h"
3 __mlu_entry__ void gemm16Kernel(half *outputDDR, half *input1DDR, half *input2DDR,
4                                 uint32_t m, uint32_t k, uint32_t n) {
5     half ret;
6     __nram__ half input1NRAM[256*256];
7     __nram__ half input2NRAM[256*256];
8     __nram__ half outputNRAM[256*256];
9     __memcpy(input1NRAM, input1DDR, m * k * sizeof(half), GDRAM2NRAM); // 从 GDRAM拷入
10    NRAM
11    __memcpy(input2NRAM, input2DDR, k * n * sizeof(half), GDRAM2NRAM);
12
13    for (uint32_t i = 0; i < m; i++) {
14        for (uint32_t j = 0; j < n; j++) {
15            ret = 0;
16            for (uint32_t t = 0; t < k; t++) {
17                ret += input1NRAM[i*k+t] * input2NRAM[t*n+j];
18            }
19            outputNRAM[i*n+j] = ret;
20        }
21    }
22    __memcpy(outputDDR, outputNRAM, m * n * sizeof(half), NRAM2GDRAM); // 将计算结果拷回
23    GDRAM
}

```

图 5.30 标量操作实现矩阵乘法的示例代码

5.2.5.3 张量操作实现

在上一步基础上，该步骤中使用 BCL 提供的向量计算指令完成矩阵乘计算。向量计算指令可以更好地发挥 DLP 硬件性能优势，提升计算效率。

这里先介绍后续步骤中要解决的矩阵乘法规模。为方便读者理解，假设左矩阵规模大小为 256×256 ，右矩阵规模大小为 $256 \times N$ （由于指令计算的对齐要求， N 必须可被 256 整除，如 327680）。如图 5.31 所示：

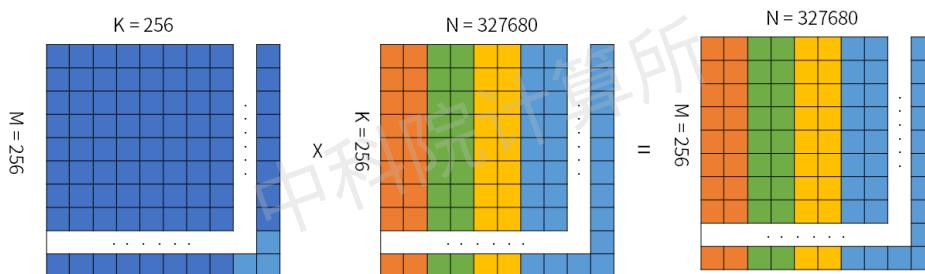


图 5.31 待解决的矩阵乘法规模

针对该问题规模，可以将输入左矩阵一次性拷入 NRAM。针对 DLP 架构特点，在执行卷积指令操作时，需要将输入的右矩阵拷入 WRAM 中，并且在向 WRAM 拷入前需要对数据进行量化处理并摆放成特定的数据格式，然后再使用 `_bang_conv` 指令进行计算。由于右矩阵规模较大，在代码中需将右矩阵分批次拷入 WRAM 进行计算。

Device 端关键代码如 5.32 所示（文件名为 gemm_CONV.mlu）。其中，all_round 表示计算的循环次数，和右矩阵规模大小相关；dst_stride 和 src_stride 代表调整右矩阵数据摆放格式过程中的步长；total_times 表示调整右矩阵数据格式需要的次数，因为实验使用的 DLP 上有 64 个卷积计算单元，需要将原本顺序摆放的数据按照 64 个为一组间隔摆放。

```

1 // filename: gemm_CONV.mlu
2 #include "mlu.h"
3 #define ROUND 256
4 __mlu_entry__ void gemm16Kernel(half *outputDDR, int8_t *input1DDR, int8_t *input2DDR,
5                                 uint32_t m, uint32_t k, uint32_t n, int16_t pos) {
6     __nram__ int8_t input1NRAM[256*256];
7     __nram__ int8_t input2NRAM[256*256];
8     __nram__ int8_t input2NRAM_tmp[256*256];
9     __wram__ int8_t input2WRAM[256*256];
10    __nram__ half outputNRAM[256*256];
11    __memcpy(input1NRAM, input1DDR, m * k * sizeof(int8_t), GDRAM2NRAM);
12    // 在这里将左矩阵一次性拷入NRAM
13
14    int all_round = n / ROUND;
15    int32_t dst_stride = (ROUND * k / 64) * sizeof(int8_t);
16    int32_t src_stride = k * sizeof(int8_t);
17    int32_t size = k * sizeof(int8_t);
18    int32_t total_times = ROUND / 64;
19    // __bang_printf("taskDim=%d, clusterId=%d, coreId=%d\n", taskDim, clusterId, coreId);
20    for(int i = 0; i < all_round; i++) {
21        __memcpy(
22            __memcpy(
23                for (int j = 0; j < total_times; j++) { // 这里将数据摆放成 bang_conv 可以使用的
24                    __memcpy(input2NRAM + j * k, input2NRAM_tmp + j * 64 * k,
25                            size, NRAM2NRAM, dst_stride, src_stride, 64);
26                }
27                __memcpy(
28                    __bang_conv(outputNRAM, input1NRAM, input2WRAM, k, m, 1, 1, 1, 1, 1, 1, ROUND, pos)
29                ;
30                for (int j = 0; j < m; j++) { // 要对每轮计算的结果进行拼接
31                    __memcpy(
32                }
33            }
34        }
35    }
36}

```

图 5.32 张量操作实现矩阵乘法的示例代码

5.2.5.4 多核并行实现

上述步骤中只调用了 DLP 的一个计算核进行计算，考虑到所使用的 DLP 有 16 个计算核，可以进一步采用 16 个计算核进行并行运算。其基本思想是根据输入矩阵规模的大小，将输入矩阵拆分成多份并分配给不同的计算核进行计算，最后再对计算结果进行合并。通过将原本由 1 个计算核承担的计算任务分配给 16 个核，大大提升了计算速度。

在这里我们对 n 进行分块并行计算。分块单位为长度 256 的数据块。每个计算核每次从 GDRAM 上拷贝数据的时候都根据自己的 CoreId 来确定目标数据的内存地址，并且只将自己负责的数据块拷入 NRAM。Device 端关键代码如 5.33 所示（文件名为 gemm_PARALL.mlu）。

```

1 // filename: gemm_PARALL.mlu
2 #include "mlu.h"
3 #define ROUND 256
4 __mlu_entry__ void gemm16Kernel(half *outputDDR, int8_t *input1DDR, int8_t *input2DDR,
5     uint32_t m, uint32_t k, uint32_t n, int16_t pos) {
6     __nram__ int8_t input1NRAM[256*256];
7     __nram__ int8_t input2NRAM[256*256];
8     __nram__ int8_t input2NRAM_tmp[256*256];
9     __wram__ int8_t input2WRAM[256*256];
10    __nram__ half outputNRAM[256*256];
11    __memcpy(input1NRAM, input1DDR, m * k * sizeof(int8_t), GDRAM2NRAM);
12        // 在这里将左矩阵一次性拷入NRAM
13
14
15    int all_round = n / ( taskDim * ROUND); // 因为现在使用16个核同时运算，所以每个核循环的次数也相应减少
16    int32_t dst_stride = (ROUND * k / 64) * sizeof(int8_t);
17    int32_t src_stride = k * sizeof(int8_t);
18    int32_t size = k * sizeof(int8_t);
19    int32_t total_times = ROUND / 64;
20
21    // __bang_printf("taskDim=%d, taskId=%d\n", taskDim, taskId);
22    for(int i = 0; i < all_round; i++) {
23        __memcpy(______); // 只涉及这个核需要的数据
24
25        for (int j = 0; j < total_times; j++) {
26            __memcpy(input2NRAM + j * k, input2NRAM_tmp + j * 64 * k,
27                     size, NRAM2NRAM, dst_stride, src_stride, 64);
28        }
29        __memcpy(______);
30        __bang_conv(______);
31        for (int j = 0; j < m; j++) { // 向GDRAM回写的时候也要注意每个核的位置不同
32            __memcpy(______);
33        }
34    }
35}

```

图 5.33 多核并行实现矩阵乘法的示例代码

5.2.5.5 SRAM 的使用

在第上一步中，因为使用了 4 个 Cluster 的 16 个计算核进行并行计算，而相同 Cluster 上的 4 个计算核在从 GDRAM 上拷贝数据到各自的 NRAM/WRAM 时，会争抢该 Cluster 到 GDRAM 的带宽，导致数据读取速度降低。考虑搭配每个 Cluster 有一个共享的 SRAM，我们将数据先从 GDRAM 拷贝到 SRAM，再从 SRAM 分发到 NRAM/WRAM 中，避免了带宽竞争问题，提高了数据读取速度。

特别注意的是，因为从 GDRAM 拷入数据到 SRAM 和从 SRAM 拷入数据到 NRAM 这两个操作是在两种不同功能的核上执行（即普通计算核和 Memory 核），所以这两个操作可以并行执行。为了保证数据一致性，需要在数据从 GDRAM 拷入到 SRAM 之后，从 SRAM 拷入到 NRAM 之前设置同步操作，即 BCL 内置的 `_sync_cluster()` 函数。其原理如 5.34 图所示。

整个执行过程如下图所示：

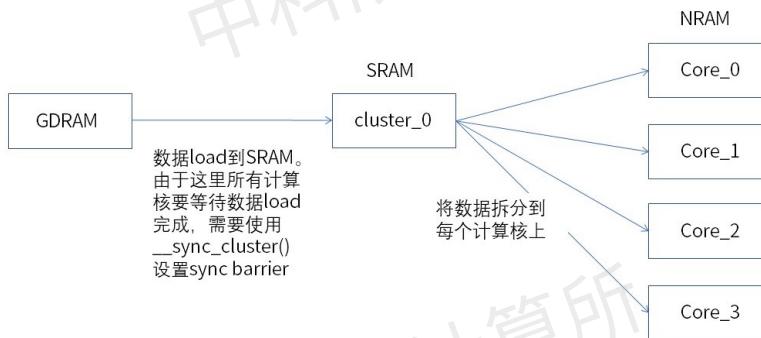


图 5.34 SRAM 的数据同步

Device 端关键代码如图5.35所示（文件名为 gemm_SRAM.mlu）。其中 clusterId 表示此时执行任务的是哪个 Cluster，范围为 [0,3]。和原本的多核并行程序相比较，带有 SRAM 的版本打破了原来各个计算核之间在时序上的独立性。由此特别需要注意在每次将数据从 GDRAM 拷贝到 SRAM 和从 SRAM 拷贝到 NRAM 都需要执行同步来保证时序上的数据一致性。其整个执行过程中的数据调度时序如图5.36所示。

5.2.5.6 访存与计算流水

由于所使用的 DLP 上有专门用于片上总线和 SRAM 管理的 SRAM 核，可以在前面步骤的基础上进一步实现访存与计算的流水。具体而言，针对 4 个并行执行的 Cluster，希望每个 Cluster 中的 SRAM 核和其他 4 个计算核构成流水线的计算模式。其中 SRAM 核只负责将数据从 GDRAM 拷入 SRAM，其余计算核则负责从 SRAM 拷入数据、完成矩阵乘法计算、将数据拷回 GDRAM。

Device 端关键代码如图访存与计算流水的示例代码所示(文件名为 gemm_PIPELINE.mlu)。其中，我们设置了在 SRAM 上的两个变量 input2SRAM1, input2SRAM2。初始时，SRAM 核从 GDRAM 上拷入数据到 input2SRAM1，当数据拷入完成后，4 个计算核开始工作，它们将自己需要的数据从 input2SRAM1 拷入进行计算。在计算核工作的同时，SRAM 核不会停止工作，它会将下一次需要计算的数据从 GDRAM 拷入 input2SRAM2，供给 4 个计算核在下一次使用，减少了计算核下一次的等待时间，input2SRAM1 和 input2SRAM2 交替读写重复上述过程直至所有数据计算完成。具体的数据调度时序如图 访存与计算流水的数据调度时序所示，从中可以发现耗时很长的从 GDRAM 到 SRAM 的数据拷贝被“隐藏”了。和之前步骤相比在相同的时间内，每次搬运了更多的 GDRAM 数据到片上并完成了计算。这也是计算机系统优化时常用到的数据流控制技巧，即“乒乓操作”。

```

1 // filename: gemm_PIPELINE.mlu
2 #include "mlu.h"
3 #define ROUND 256
4 __mlu_entry__ void gemm16Kernel(half *outputDDR, int8_t *input1DDR, int8_t *input2DDR,
5     uint32_t m, uint32_t k, uint32_t n, int16_t pos) {
6     __nram__ int8_t input1NRAM[256*256];
7 }
```

```

1 // filename: gemm_SRAM.mlu
2 #include "mlu.h"
3 #define ROUND 256
4 __mlu_entry__ void gemm16Kernel(half *outputDDR, int8_t *input1DDR, int8_t *input2DDR,
5     uint32_t m, uint32_t k, uint32_t n, int16_t pos) {
6     __nram__ int8_t input1NRAM[256*256];
7     __nram__ int8_t input2NRAM[256*256];
8     __nram__ int8_t input2NRAM_tmp[256*256];
9     __wram__ int8_t input2WRAM[256*256];
10    __nram__ half outputNRAM[256*256];
11    __memcpy(input1NRAM, input1DDR, m * k * sizeof(int8_t), GDRAM2NRAM);
12    //在这里将左矩阵一次性拷入NRAM
13    int all_round = n / (taskDim * ROUND); // 因为现在使用16个核同时运算，所以每个核循环的次数也相应减少
14    int32_t dst_stride = (ROUND * k / 64) * sizeof(int8_t);
15    int32_t src_stride = k * sizeof(int8_t);
16    int32_t size = k * sizeof(int8_t);
17    int32_t total_times = ROUND / 64;
18    __mlu_shared__ int8_t input2SRAM[256*1024];
19    //_bang_printf("taskDim=%d, clusterId=%d, coreId=%d\n", taskDim, clusterId, coreId);
20    for(int i = 0; i < all_round; i++) {
21    {
22        // copy GDRAM2SRAM
23        __memcpy(______); // 只将右矩阵拷入SRAM中
24        __sync_cluster(); //设置sync barrier
25        // copy SRAM2NRAM
26        __memcpy(______);
27
28        // 将数据摆好对应的格式
29        for (int j = 0; j < total_times; j++) {
30            __memcpy(input2NRAM + j * k, input2NRAM_tmp + j * 64 * k,
31                     size, NRAM2NRAM, dst_stride, src_stride, 64);
32        }
33        // copy NRAM2WRAM
34        __memcpy(______);
35        // compute
36        __bang_conv(______);
37        // copy NRAM2GDRAM
38        for (int j = 0; j < m; j++) { // 向GDRAM回写的时候也要注意每个核的位置不同
39            __memcpy(______);
40        }
41    }
42    __sync_cluster(); //设置sync barrier
43 }
44 }
```

图 5.35 使用 SRAM 的示例代码

	时间片 t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	
mem core	数据拷贝 GDRAM 到 SRAM				数据拷贝 GDRAM 到 SRAM				数据拷贝 GDRAM 到 SRAM		
core0		数据拷贝 SRAM 到 NRAM	张量计算	数据拷贝 NRAM 到 GDRAM		数据拷贝 SRAM 到 NRAM	张量计算	数据拷贝 NRAM 到 GDRAM		数据拷贝 SRAM 到 NRAM	
core1		数据拷贝 SRAM 到 NRAM	张量计算	数据拷贝 NRAM 到 GDRAM		数据拷贝 SRAM 到 NRAM	张量计算	数据拷贝 NRAM 到 GDRAM		数据拷贝 SRAM 到 NRAM
core2		数据拷贝 SRAM 到 NRAM	张量计算	数据拷贝 NRAM 到 GDRAM		数据拷贝 SRAM 到 NRAM	张量计算	数据拷贝 NRAM 到 GDRAM		数据拷贝 SRAM 到 NRAM	
core3		数据拷贝 SRAM 到 NRAM	张量计算	数据拷贝 NRAM 到 GDRAM		数据拷贝 SRAM 到 NRAM	张量计算	数据拷贝 NRAM 到 GDRAM		数据拷贝 SRAM 到 NRAM	
	第一块数据处理			第二块数据处理			第三块数据处理.....				

图 5.36 SRAM 的数据调度时序

```

8    __nrnram__ int8_t input2NRAM[256*256];
9    __nrnram__ int8_t input2NRAM_tmp[256*256];
10   __wram__  int8_t input2WRAM[256*256];
11   __nrnram__ half outputNRAM[256*256];
12   __memcpy(input1NRAM, input1DDR, m * k * sizeof(int8_t), GDRAM2NRAM);
13   // 在这里将左矩阵一次性拷入NRAM
14   int all_round = n / (taskDim * ROUND); // 因为现在使用16个核同时运算，所以每个核循环的次数也
15   // 相应减少
16   int32_t dst_stride = (ROUND * k / 64) * sizeof(int8_t);
17   int32_t src_stride = k * sizeof(int8_t);
18   int32_t size = k * sizeof(int8_t);
19   int32_t total_times = ROUND / 64;
20   __mlu_shared__ int8_t input2SRAM1[256*1024];
21   __mlu_shared__ int8_t input2SRAM2[256*1024];
22   __mlu_shared__ int8_t * input2SRAM_read;
23   __mlu_shared__ int8_t * input2SRAM_write;
24   input2SRAM_write=input2SRAM1;
25   // copy GDRAM2SRAM
26   __memcpy(input2SRAM_write, input2DDR + ROUND * (clusterId * 4) * k,
27             k * ROUND * 4 * sizeof(int8_t), GDRAM2SRAM); // 只将右矩阵拷入SRAM中
28   __sync_cluster(); // 设置 sync barrier
29   // __bang_printf("taskDim=%d, clusterId=%d, coreId=%d\n", taskDim, clusterId, coreId);
30   for(int i = 0; i < all_round-1; i++)
31   {
32       if (i % 2 == 0)
33       {
34           input2SRAM_read=input2SRAM1;
35           input2SRAM_write=input2SRAM2;
36       } else
37       {
38           input2SRAM_read=input2SRAM2;
39           input2SRAM_write=input2SRAM1;
40       }
41       // copy GDRAM2SRAM
42       __memcpy(______________________); // 只将右矩阵拷入SRAM中
43       // copy SRAM2NRAM
44       __memcpy(______________________);

```

```

45 // 将数据摆好对应的格式
46 for (int j = 0; j < total_times; j++) {
47     __memcpy(input2NRAM + j * k, input2NRAM_tmp + j * 64 * k,
48             size, NRAM2NRAM, dst_stride, src_stride, 64);
49 }
50 // copy NRAM2WRAM
51 __memcpy(input2WRAM, input2NRAM, ROUND*k*sizeof(int8_t), NRAM2WRAM);
52 // compute
53 __bang_conv(_____);
54 // copy NRAM2GDRAM
55 for (int j = 0; j < m; j++) { // 向GDRAM回写的时候也要注意
    每个核的位置不同
        __memcpy(_____);
    }
56     __sync_cluster(); // 设置sync barrier
57 }
58 __memcpy(_____);
59
60 // 将数据摆好对应的格式
61 for (int j = 0; j < total_times; j++) {
62     __memcpy(_____);
63 }
64 // copy NRAM2WRAM
65 __memcpy(_____);
66 // compute
67 __bang_conv(_____);
68 // copy NRAM2GDRAM
69 for (int j = 0; j < m; j++) { // 向GDRAM回写的时候也要注意每个
    核的位置不同
        __memcpy(_____);
    }
70 }
71 }
72 }
73 }
74 }

```

Listing 2 访存与计算流水的示例代码

	时间片 t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	
mem core	数据拷贝 GDRAM到 SRAM(S1)	数据拷贝 GDRAM到 SRAM(S2)			数据拷贝 GDRAM到 SRAM(S1)			数据拷贝 GDRAM到 SRAM(S2)			
core0		数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S2)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	
core1		数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S2)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	...
core2		数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S2)	张量计算	数据拷贝 NRAM到 GDRAM	数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDRAM	
core3		数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDR	数据拷贝 SRAM到 NRAM(S2)	张量计算	数据拷贝 NRAM到 GDR	数据拷贝 SRAM到 NRAM(S1)	张量计算	数据拷贝 NRAM到 GDR	
	第一块数据处理				第二块数据处理			第三块数据处理.....			

图 5.37 访存与计算流水的数据调度时序

5.2.6 实验评估

本实验主要考虑用智能编程语言解决特定问题，实验评估指标主要为程序本身对问题规模的支持范围和性能。本实验设定的评估标准如下：

- 60 分标准：在规模 $m=256, k=256, n=327680$ 下 DLP 计算结果与 CPU 计算结果误差在 10% 以内。
- 80 分标准：在规模 $m=256, k=256, n=327680$ 下 DLP 计算结果与 CPU 计算结果误差在 1% 以内。
- 90 分标准：在规模 $m=256, k=256, n=327680$ 下 DLP 计算结果与 CPU 计算结果误差在 0.1% 以内，耗时小于 20ms。
- 100 分标准：在规模 $m=256, k=256, n=327680$ 下 DLP 计算结果与 CPU 计算结果误差在 0.06% 以内，耗时小于 15ms。

5.2.7 实验思考

1. _bang_conv 指令对数据摆放有特殊要求的原因是什么？
2. 在 SRAM 的使用代码 5.35 中，其中有两个 sync，这是为什么？
3. 以上程序目前的瓶颈在哪里？有什么方法可以验证？
4. 访存与计算流水的设计是否还有其他方案？是否有可能从整个软硬件系统的角度进一步提升性能？
5. 加入输入数据不是 256 的整数倍，程序该怎么修改以适应这种变化？

第6章 * 深度学习处理器运算器设计

随着深度学习应用场景越来越复杂，深度学习算法的运算形式日益多样化、网络结构愈加复杂，例如 Alexnet、GoogleNet、ResNet 等。为了更高效、更灵活地支持深度学习算法，处理器设计需要针对深度学习算法的特性进行优化加速以满足编程者越来越多样化的需求。深度学习加速器就是一类针对高效支持深度学习算法的处理器，其针对深度学习的通用计算进行加速，例如卷积运算、池化运算等；同时，深度学习加速器考虑深度学习算法算子多样性，提供灵活的指令集，便于程序员高效地实现算法。

在深度学习算法中，卷积运算是最核心的运算操作，卷积层包含大量的输入输出数据和权值参数，其运算量占深度学习算法总运算量的 90% 以上。处理器执行卷积运算的性能也决定了深度学习算法在处理器上的性能。在智能处理器系统中，设计出更高效地支持卷积运算的运算器，是深度学习加速器设计的关键技术之一。

6.1 实验目的

在本章节中，将根据深度学习加速器卷积运算原理，使用 Verilog HDL 编写程序设计出深度学习处理器中矩阵运算子单元——内积运算器，然后在 Mentor 公司的 Modelsim 环境下用 Verilog 进行仿真。这个内积运算器是一个专门为教学目的简化的设计。在此设计过程中，我们不但关注组成内积运算器设计的每个模块是可仿真并且可以综合成门级网表，而且关心内积运算器的性能。因此，这个设计是一个能真正通过具体物理电路实现的内积运算器。本章实验目的的具体包括：

- 1) 学习深度学习处理器矩阵运算器加速卷积层原理，理解本实验和深度学习处理器基本模块间的关系。
- 2) 完成串行内积运算器，初步理解内积运算器的基本组成单元。
- 3) 结合对串行内积运算器的认识和深度学习处理器中矩阵运算器加速原理完成并行内积运算器，加深对深度学习处理器加速卷积计算原理的理解。
- 4) 在并行内积运算器基础上完成矩阵运算子单元，加深对矩阵运算子单元的理解。

实验工作量：约需 4 个小时

本章节的实验设计仅仅是一个教学模型，从物理实现上来看结构设计不一定很合理，主要是为了从原理上说明深度学习处理器中矩阵运算是如何加速卷积计算。

6.2 背景介绍

本节首先介绍分析深度学习算法中卷积层的算法特征，然后介绍面向卷积运算的深度学习处理器架构，最后介绍矩阵运算以及卷积层在深度学习处理器上的处理过程。

6.2.1 卷积层算法特征

卷积层由输入层、输出层和若干个卷积核 (filter) 组成。输入层和输出层包括若干个特征图像。每个卷积核包含 $K_x \times K_y$ 组权值，其与输入层 $K_x \times K_y$ 区域内神经元对位相乘后累加乘法结果即可得到一个输出神经元。卷积核沿着输入特征图像的 X、Y 方向以 S_x, S_y 的步长滑动遍历则得到一个输出特征图像，然后遍历所有卷积核得到输出层所有特征图像。

根据卷积算法，卷积计算可先采用输入层 $K_x \times K_y$ 区域内神经元和所有的卷积核计算得到不同输出特征图像上相同位置的输出神经元，然后 $K_x \times K_y$ 区域框沿着输入特征图像的 X、Y 方向滑动遍历得到输出特征图像的所有神经元。第 fo 个特征图像上 (w, h) 位置的输出神经元计算公式为：

$$OUT_{w,h}^{fo} = f\left(\sum_{i=0}^{K_x-1} \sum_{j=0}^{K_y-1} \sum_{fi=0}^{F_i-1} W_{fi,i,j}^{fo} * IN_{wS_x+i, hS_y+j}^{fi}\right) + \beta^{fo} \quad (6.1)$$

其中， W 、 IN 、 OUT 、 β 分别表示权值、输入神经元、输出神经元、偏置， f 表示激活函数。

公式6.1可等效为：

$$OUT_{w,h;fo} = f\left(\sum_{i=0}^{F_i * K_x * K_y - 1} \bar{W}_{fo,i} * \bar{IN}_{i,w,h} + \beta^{fo}\right) \quad (6.2)$$

其中， $\bar{IN}_{i,w,h}$ 是一个包含 $F_i * K_x * K_y$ 个分量的向量，由输入层中对应计算 (w, h) 位置的输出结果的 $K_x \times K_y$ 区域内神经元组成； \bar{W} 为权值矩阵，其规模为 $fo \times (F_i * K_x * K_y)$ ，由卷积层 $fo \times K_y \times K_x \times F_i$ 规模的 4 维权值张量的低三个维度合并成一个维度转换而来。

由公式6.2可知，卷积计算可以看成先进行 $W * H$ 次 $F_i * K_x * K_y$ 分量的输入神经元向量和 $(F_i * K_x * K_y) \times fo$ 规模的权值矩阵相乘，然后进行向量加法和向量激活。

另一方面，当规模为 m 的向量 A 和规模为 $m \times n$ 的矩阵 B 相乘时，得到的乘积向量 C 的第 i 个元素可表示为：

$$c_i = \sum_{j=0}^{m-1} a_j * b_{j,i} \quad (6.3)$$

公式6.3可表示为，

$$c_i = A \cdot B_i \quad (6.4)$$

即乘积结果向量 C 的第 i 个元素为向量 A 和矩阵 B 的第 i 个列向量 B_i 的内积。

结合式6.2和式6.4可知，卷积运算有一系列向量内积运算、向量加法和向量激活组成，尤其以向量内积运算为主。

6.2.2 面向卷积运算的 DLP 架构

在图6.1所示由控制模块、运算模块和存储模块三大部分组成的智能计算系统中深度学习处理器架构 (Deep Learning Processor, DLP) 中，输入/输出神经元、权值分别存储于神经

元存储单元（Neuron RAM, NRAM）和权值存储单元（Weight RAM, WRAM）；矩阵乘运算由矩阵运算单元（Matrix Function Unit, MFU）完成；其他向量运算由向量运算单元（Vector Function Unit, VFU）完成。控制模块则负责协调控制运算模块和存储模块完成深度学习任务。

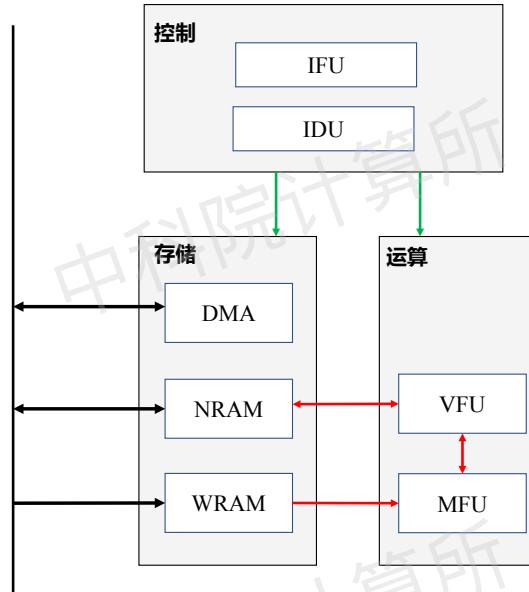


图 6.1 深度学习处理器结构图

MFU 通过图6.2所示的 H 树的互联方式将 M 个矩阵运算子单元（Processing Element, PE）组织为一个完整的矩阵运算单元，不同 PE 位于 H 树的叶节点。H 树将预处理后的输入神经元和控制信号广播到所有 PE，并收集不同 PE 计算的输出结果返回给 VFU。PE 单元负责进行向量的内积运算，主要由 N 个乘法器和一个 N 输入加法树组成。

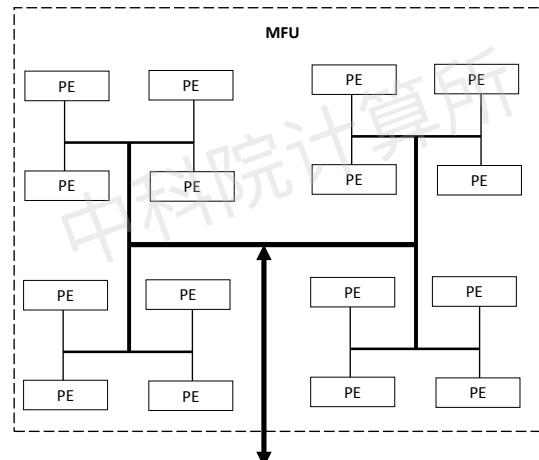


图 6.2 矩阵运算单元结构图

类似 MFU, WRAM 也采用 H 树互连的方式将 M 个分布式的片上存储单元（Distributed

Weight RAM, DWRAM) 组织在一起。在深度学习处理器中，每个 DWRAM 对应一个 PE。矩阵运算时，每个 DWRAM 根据控制信号读取权值信号给 PE 进行计算。

6.2.3 DLP 上矩阵及卷积处理过程

深度学习处理器进行规模为 m 的向量 A 和规模为 $m \times n$ 的矩阵 B 相乘时，将矩阵 B 均分为 M 个子矩阵 $\bar{B}_0, \bar{B}_1, \dots, \bar{B}_{M-1}$ 。每个子矩阵规模为 $m \times \frac{n}{M}$ ，分别存储在 M 个 DWRAM 中。MFU 进行计算时，MFU 的 M 个 PE 利用接收 H 树广播的向量 A 的 N 个分量，并分别从 M 个 DWRAM 读取各自子矩阵的第 i 个列向量的 N 个分量进行内积运算，得到输出向量 C 的 M 个分量的部分和。处理器控制 NRAM 将向量 A 的所有分量都发送给 MFU 则可完成输出向量 C 的 M 个分量的计算。然后，处理器将前面步骤重复 $\frac{n}{M}$ 次，便可完成输出向量 C 所有分量的计算。

假设矩阵运算单元 (MFU) 包含 4 个矩阵运算子单元 (PE)，输入的神经元矩阵 A 规模为 2×8 、权值矩阵 B 规模为 8×4 ，计算的输出神经元 C 规模是 2×4 。矩阵运算单元进行 $C = A \times B$ 运算时，每个矩阵运算子单元计算 C 的列子矩阵。如图6.3所示，矩阵运算单元需 4 cycle 完成矩阵运算：

- 第 1 拍，第一行的前 4 个神经元广播给所有 PE，每个 PE 接收对应列权值的前 4 个元素，分别进行内积运算，得到第一行 4 个输出神经元结果的部分和。
- 第 2 拍，第一行的后 4 个神经元广播给所有 PE，每个 PE 接收对应列权值的后 4 个元素，分别进行内积运算，然后累加第 1 拍计算的部分和结果，得到第一行 4 个输出神经元。
- 第 3 拍，第二行的前 4 个神经元广播给所有 PE，每个 PE 接收对应列权值的前 4 个元素，分别进行内积运算，得到第二行 4 个输出神经元结果的部分和。
- 第 4 拍，第二行的后 4 个神经元广播给所有 PE，每个 PE 接收对应列权值的后 4 个元素，分别进行内积运算，然后累加第 3 拍计算的部分和结果，得到第二行 4 个输出神经元。

卷积计算时，DLP 将卷积层的所有输出特征图像 (Output Feature Map, OFM) 以 M 个特征图像为一组。如图6.4所示，DLP 每次采用输入神经元中 $K_x \times K_y \times F_i$ 数据子块计算一组输出特征图像中不同输出特征图像上相同位置的特征点。MFU 的 M 个矩阵运算子单元分别计算不同输出特征图像的输出神经元。然后，DLP 按照 XY 循环顺序计算特征图像上的特征点，计算一组完整的输出特征图像。

DLP 计算一组输出特征图像中不同输出特征图像上相同位置的特征点过程类似于矩阵运算。运算过程可拆分为 4 个步骤：

- 步骤一：VFU 依次将 $K_x \times K_y \times F_i$ 数据子块从 NRAM 读出，进行数据预处理后发送给 MFU。
- 步骤二：MFU 将接收的输入神经元广播给 M 个 PE 单元。
- 步骤三：PE 接收 H 树广播的输入神经元和 WRAM 读取权值进行内积运算，并将内积结果保存至部分和缓存寄存器或将内积结果累加部分和缓存器的数值。
- 步骤四：PE 收到 $K_x \times K_y \times F_i$ 数据子块的所有数据后将计算的部分和进行数据格式

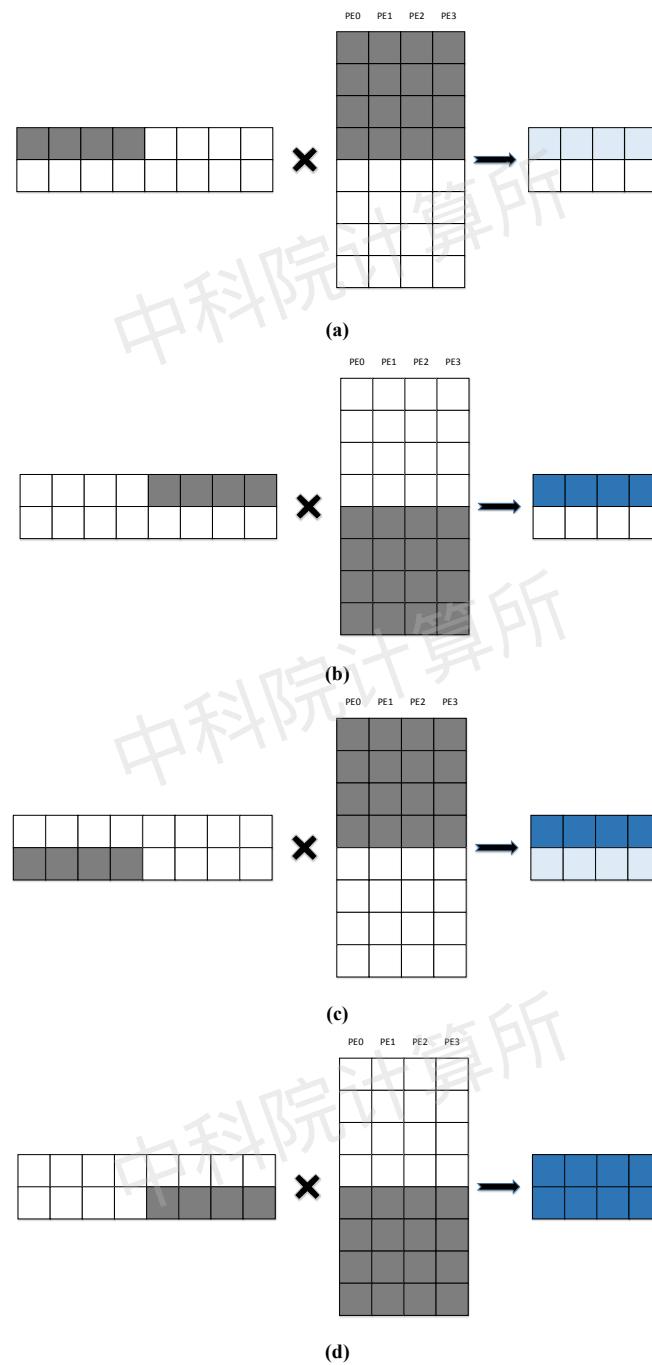


图 6.3 4PE 矩阵运算单元矩阵乘步骤示意图

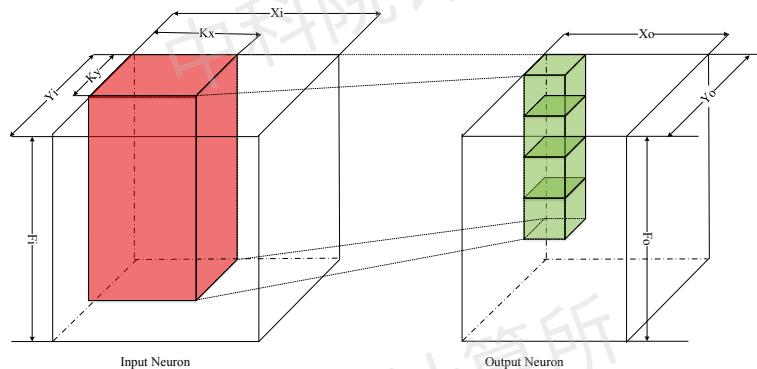


图 6.4 卷积层计算示意图

转换后输出。

通过矩阵运算过程和卷积运算过程可知，矩阵运算子单元的内积运算是矩阵运算单元的核心。下面将详细介绍内积运算单元设计和仿真过程。

6.3 实验内容

根据第6.2节可知，矩阵运算子单元的内积运算器是深度学习处理器的核心部件。本节将通过两个实验介绍矩阵运算子单元的不同实现方法。然后通过对比两个实验效果阐述深度学习处理器加速深度学习算法内积运算的原理。

矩阵运算子单元的内积运算器的功能是将长度可变的神经元向量（neuron）和权值向量（weight）进行内积运算，然后将结果输出。其功能伪代码如图6.5所示。

```

1 psum = 0;
2 for(i = 0; i < element_num; i++){
3     psum = neuron[i] * weight[i];
4 }
5 output = psum;

```

图 6.5 内积运算器功能代码

在通用 CPU 中，深度学习神经元数据和权值数据一般采用单精度浮点数据表示，相应的运算单元也采用的浮点运算器。然而在深度学习处理器中，为了节省功耗、面积开销，一般采用低精度运算器代替浮点运算器。根据文献^[3]所述，INT16 或者 INT8 已经能够满足深度学习算法的应用需求。为了简化实验内容，本章实现的内积运算所有神经元/权值数据都采用 INT16 表示。

为了方便逐步理解深度学习矩阵运算子单元设计和迭代开发，本实验分为逐步递进的三个步骤：

- (1) 串行内积运算器设计，初步理解内积运算器的基本组成单元。

(2) 并行内积运算器设计，完成矩阵运算子单元基本运算单元。同时，加深对深度学习处理器加速卷积计算原理的理解。

(3) 矩阵运算子单元设计，在并行内积运算器基础上增加控制逻辑完成矩阵运算子单元，加深对矩阵运算子单元的理解。

6.4 实验步骤

6.4.1 串行内积运算器

串行内积运算器每拍最多接收一个神经元和一个权值分量进行乘法运算，然后再将乘法结果累加到部分和寄存器。当串行内积运算器处理的神经元/权值数据是一组神经元/权值向量的第一个元素时，乘法结果直接写入部分和寄存器，不需进行累加；当处理的神经元/权值数据是一组神经元/权值向量的最后一个元素时，串行内积运算器将累加结果输出。

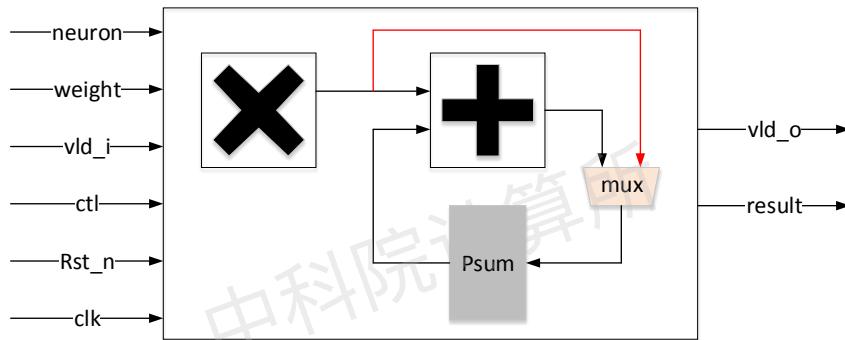


图 6.6 串行内积运算器

串行内积运算器的结构如图6.6所示，主要包括：一个乘法器、一个加法器、一个部分和寄存器和一个数据选择器。其输入输出接口信号描述如表6.1所示。其中，控制信号 **ctl** 为 2bit 信号，最低位表示输入神经元/权值数据是一组神经元/权值向量的第一个元素。当 **ctl[0]** 有效时，乘法结果直接写入部分和寄存器；否则，乘法结果先累加部分和寄存器，然后将累加结果写入部分和寄存器。图6.6中数据选择器即用于选择写入部分和寄存器的源数据。控制信号 **ctl** 的最高位表示输入神经元/权值数据是一组神经元/权值向量的最后一个元素。当 **ctl[1]** 有效时，串行内积运算器在下一个时钟周期将部分和寄存器输出到 **result** 端口。串行内积运算器输出 **result** 值，**vld_o** 置起 1 拍，表示输出内积结果有效。

串行内积运算器采用异步复位方式，复位信号 **rst_n** 低电平有效。复位时，部分和寄存器被清零。输入 **vld_i** 表示输入表示神经元/权值数据和控制信号有效，当 **vld_i** 为高电平时，串行内积运算器接收输入的 INT16 神经元和 INT16 权值分量进行乘法运算，然后再用乘法结果更新部分和寄存器。串行内积运算器代码示例如图6.7所示。

乘法单元输入的神经元向量和权值向量中每个神经元和权值数据都是有符号数据，乘法后得到的部分和也是有符号数据，部分和数据位宽为对应神经元位宽与权值位宽之和。Verilog 语法中乘法运算符默认进行无符号乘法运算，有符号数据运算需进行显示说明。

表 6.1 串行内积运算器信号描述

域	位宽	功能描述
neuron	16	输入 INT16 神经元分量
weight	16	输入 INT16 权值分量
vld_i	1	输入数据和控制信号有效标志, 高电平有效
ctl	2	输入控制信号 ctl[0]: 输入神经元/权值数据是一组神经元/权值向量的第一个元素 ctl[1]: 输入神经元/权值数据是一组神经元/权值向量的最后一个元素
rst_n	1	输入复位信号, 低电平有效
clk	1	输入时钟信号
result	32	输出内积结果
vld_o	1	输出内积结果有效标志, 高电平有效

```

1 /* file: serial_pe.v*/
2 module serial_pe(
3   input          clk ,
4   input          rst_n ,
5   input signed [15:0] neuron ,
6   input signed [15:0] weight ,
7   input          [ 1:0] ctl ,
8   input          vld_i ,
9   output         [31:0] result ,
10  output reg     vld_o
11 );
12 /* multiplier*/
13 wire signed [31:0] mult_res = /*TODO*/;
14 reg [31:0] psum_r;
15
16 /* adder*/
17 wire [31:0] psum_d = /*TODO*/;
18
19 /* partial sum reg*/
20 always@(posedge clk or negedge rst_n)
21 if(!rst_n) begin
22   psum_r <= 32'h0;
23 end else if(vld_i) begin
24   psum_r <= psum_d;
25 end
26
27 always@(posedge clk or negedge rst_n)
28 if(!rst_n) begin
29   vld_o <= 1'b0;
30 end else if(/*TODO*/) begin
31   vld_o <= 1'b1;
32 end else begin
33   vld_o <= 1'b0;
34 end
35 endmodule

```

图 6.7 串行内积运算器代码

6.4.2 并行内积运算器

不同于串行内积运算器每拍最多处理一个神经元和一个权值分量进行乘法运算，并行内积运算器每拍能处理多个神经元/权值分量。并行内积运算器的结构示意图如图6.8所示，包含一组（32个）乘法器、一个累加单元、一个加法器、一个部分和寄存器和一个数据选择器。

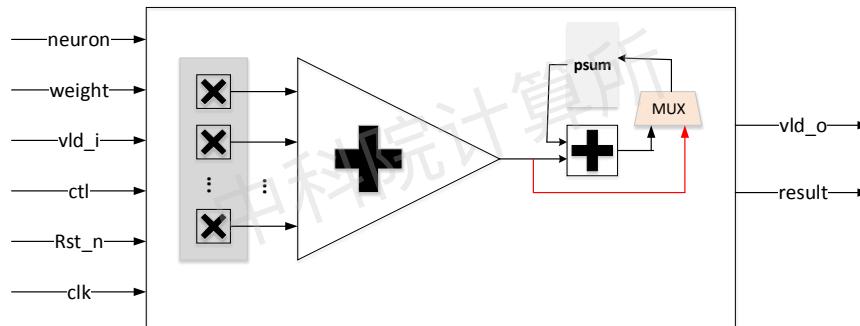


图 6.8 并行内积运算器

并行内积运算器接口信号如表6.2所示，与串行内积运算器接口信号类似。不同之处在于，并行内积运算器输入的神经元数据和权值数据为32个INT16分量的向量，而串行内积运算器输入的神经元数据和权值数据是1个INT16分量。

表 6.2 并行内积运算器信号描述

域	位宽	功能描述
neuron	512	输入神经元数据，包括32个INT16分量
weight	512	输入权值数据，包括32个INT16分量
vld_i	1	输入数据和控制信号有效标志，高电平有效
ctl	2	输入控制信号 ctl[0]: 输入神经元/权值数据是一组神经元/权值向量的第一个元素 ctl[1]: 输入神经元/权值数据是一组神经元/权值向量的最后一个元素
rst_n	1	输入复位信号，低电平有效
clk	1	输入时钟信号
result	32	输出内积结果
vld_o	1	输出内积结果有效标志，高电平有效

并行内积运算器每次进行包含32个INT16分量的神经元向量和权值向量内积，然后通过将连续32个INT16分量的内积结果累加来支持更长神经元向量和权值向量的内积运算。当vld_i为高电平时，并行内积运算器接收输入的INT16神经元和INT16权值向量，将不同的神经元分量和权值分量输入对应的乘法器进行乘法运算。然后，将32个乘法器的结果输出给累加单元。32个并行乘法器代码如图6.9所示。

累加单元将32个输入部分和累加成一个部分和，其代码如图6.10所示。

如图6.11所示，并行内积运算器将并行乘法器和累加单元。然后，通过控制信号将并行累加器的结果和部分和寄存器进行累加，并生成输出结果有效信号：

当ctl[0]有效时，累加单元结果直接写入部分和寄存器；否则，累加单元结果先通过加

```
1 /* file: pe_mult.v*/
2 module pe_mult(
3     input [ 511:0] mult_neuron,
4     input [ 511:0] mult_weight,
5     output [1023:0] mult_result
6 );
7
8 /* int16 mult */
9 genvar i;
10 wire signed [15:0] int16_neuron[31:0];
11 wire signed [15:0] int16_weight[31:0];
12 wire signed [31:0] int16_mult_result[31:0];
13 generate
14     for(i=0; i<32; i=i+1)
15         begin:int16_mult
16             /* TODO */
17         end
18     endgenerate
19
20 endmodule
```

图 6.9 并行乘法运算器代码

```
1 /* file: pe_acc.v*/
2 module pe_acc(
3     input [1023:0] mult_result,
4     output [ 31:0] acc_result
5 );
6 genvar i;
7 genvar j;
8
9 /* int16 add tree */
10 wire [31:0] int16_result[31:0][5:0];
11 for(i=0; i<=5; i=i+1)
12     begin:int16_add_tree
13         for(j=0; j<32/(2**i); j=j+1)
14             begin:int16_adder
15                 if(i==0) begin
16                     assign int16_result[0][j] = /*TODO */;
17                 end else begin
18                     assign int16_result[i][j] = /* TODO */;
19                 end
20             end
21     end
22 assign acc_result = int16_result[5][0];
23 endmodule
```

图 6.10 累加单元代码

法器累加部分和寄存器，然后将累加结果写入部分和寄存器。图6.8中数据选择器即用于选择写入部分和寄存器的源数据。

控制信号最高位 `ctl[1]` 表示输入神经元/权值数据是一组神经元/权值向量的最后一个子向量。当 `ctl[1]` 有效时，并行内积运算器在下一个时钟周期将部分和寄存器输出到 `result` 端口。并行内积运算器输出 `result` 值，`vld_o` 置起 1 拍，表示输出内积结果有效。

```

1  /* file: parallel_pe.v*/
2  module parallel_pe(
3      input          clk ,
4      input          rst_n ,
5      input [511:0]  neuron ,
6      input [511:0]  weight ,
7      input [ 1:0]   ctl ,
8      input          vld_i ,
9      output [ 31:0] result ,
10     output reg    vld_o
11 );
12 wire [511:0] mult_result;
13 pe_mult u_pe_mult(
14     .mult_neuron (neuron),
15     .mult_weight (weight),
16     .mult_result (mult_result)
17 );
18
19 wire [31:0] acc_result;
20 pe_acc u_pe_acc(
21     .mult_result (mult_result),
22     .acc_result  (acc_result)
23 );
24
25 reg [31:0] psum_r;
26 wire [31:0] psum_d = /*TODO*/;
27
28 always@(posedge clk or negedge rst_n)
29 if(!rst_n) begin
30     psum_r <= 32'h0;
31 end else if(vld_i) begin
32     psum_r <= psum_d;
33 end
34
35 always@(posedge clk or negedge rst_n)
36 if(!rst_n) begin
37     vld_o <= 1'b0;
38 end else if(/*TODO*/) begin
39     vld_o <= 1'b1;
40 end else begin
41     vld_o <= 1'b0;
42 end
43 endmodule

```

图 6.11 并行内积运算器代码

6.4.3 矩阵运算子单元

根据第6.2.2节的描述，矩阵运算子单元根据控制信号来接收 H 树广播的神经元向量和 WRAM 读取的权值向量数据进行内积运算，其结构如图6.12所示。由于矩阵运算子单元和 H 树总线、WRAM 相连接，对应神经元、权值、输出结果的接口信号变成带有反压 ready 握手的总线信号。相对于并行内积运算器，矩阵运算子单元增加了控制单元（CTL），并增加了控制数据、控制信号输入或输出的控制单元。

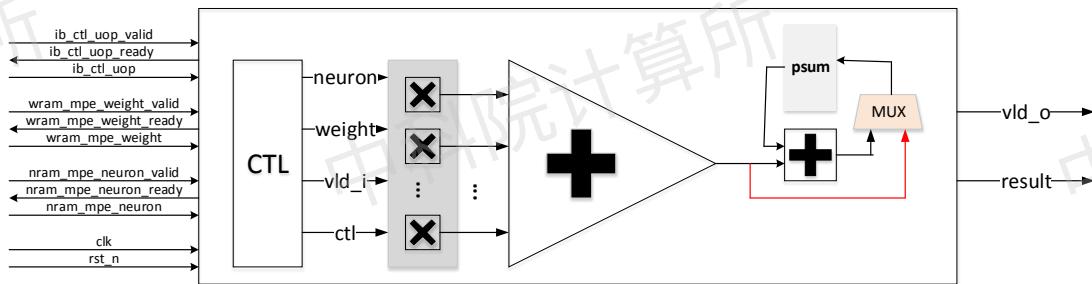


图 6.12 矩阵运算子单元

矩阵运算子单元的代码如图6.13所示。

控制单元接收输入的控制信号（**uop**），然后根据控制信号进一步译码生成输出给乘法单元的控制信号。矩阵运算子单元接收的每条控制信号对应计算一个输出神经元。每个输出神经元采用 k 次循环控制输入神经元和权值向量的乘累加。即一条控制信号计算的输出神经元需进行 k 次乘累加。相应地，控制单元也将生成出 k 条给乘法单元的控制信号。

输入控制单元的控制信号（**uop**）为 8 比特位宽的信号，表示对应计算的输出神经元输入内积神经元和权值的长度（单位：64 Byte）。控制单元根据控制信号译码时，生成给乘法单元的第一条 **ctl** 信号中 **ctl[0]** 为 1，表示对应的神经元和权重内积结果直接保存至部分和寄存器，不需累加部分和寄存器；最后一条 **ctl** 信号中 **ctl[1]** 为 1，表示最后一组部分和计算完成，可以将部分和结果输出。当一条输入控制信号译码生成的所有 **ctl** 信号输出后，控制单元可切换下一条控制信号进行译码，输出 **ib_ctl_uop_ready** 信号变为高电平。

同时，控制单元接收输入的权值数据和神经元数据，与对应给乘法单元的控制信号同步后一起输出。仅当输入的神经元数据、权值数据和控制信号都有效时，控制单元才会将这三组数据发送给乘法单元，并完成输入神经元/权值数据的握手。

```

1  /* file: matrix_pe.v*/
2  module matrix_pe(
3      input          clk ,
4      input          rst_n ,
5      input [511:0] nram_mpe_neuron ,
6      input          nram_mpe_neuron_valid ,
7      output         nram_mpe_neuron_ready ,
8      input [511:0] wram_mpe_weight ,
9      input          wram_mpe_weight_valid ,
10     output        wram_mpe_weight_ready ,
11     input [ 7:0] ib_ctl_uop ,
12     input          ib_ctl_uop_valid ,
13     output reg    ib_ctl_uop_ready ,
14     output [ 31:0] result ,
15     output reg    vld_o
16 );
17 reg inst_vld;
18 reg [7:0] inst , iter;
19 always@(posedge clk or negedge rst_n) begin
20     /*TODO: inst_vld & inst*/
21 end
22 always@(posedge clk or negedge rst_n) begin
23     /*TODO: iter*/
24 end
25 always@(posedge clk or negedge rst_n) begin
26     /*TODO: ib_ctl_uop_ready*/
27 end
28
29 wire [1:0] pe_ctl;
30 assign pe_ctl[0] = /*TODO*/;
31 assign pe_ctl[1] = /*TODO*/;
32 wire pe_vld_i = /*TODO*/;
33 wire [31:0] pe_result;
34 wire pe_vld_o;
35 parallel_pe u_parallel_pe /*TODO*/;
36
37 assign nram_mpe_neuron_ready = /*TODO*/;
38 assign wram_mpe_weight_ready = /*TODO*/;
39 assign result = pe_result;
40 assign vld_o = pe_vld_o;
41 endmodule

```

图 6.13 并行内积运算器代码

6.5 实验环境

本节将描述实验环境，包括工具安装和验证环境搭建。

6.5.1 工具安装

本章节所有程序都在 Mentor 公司的 Modelsim 10.4a (学生版) 上运行。软件安装包可从 Mentor 官网下载页面 (https://www.mentor.com/company/higher_ed/modelsim-student-edition) 下载学生版软件。下载安装包后，按照如下步骤安装程序：

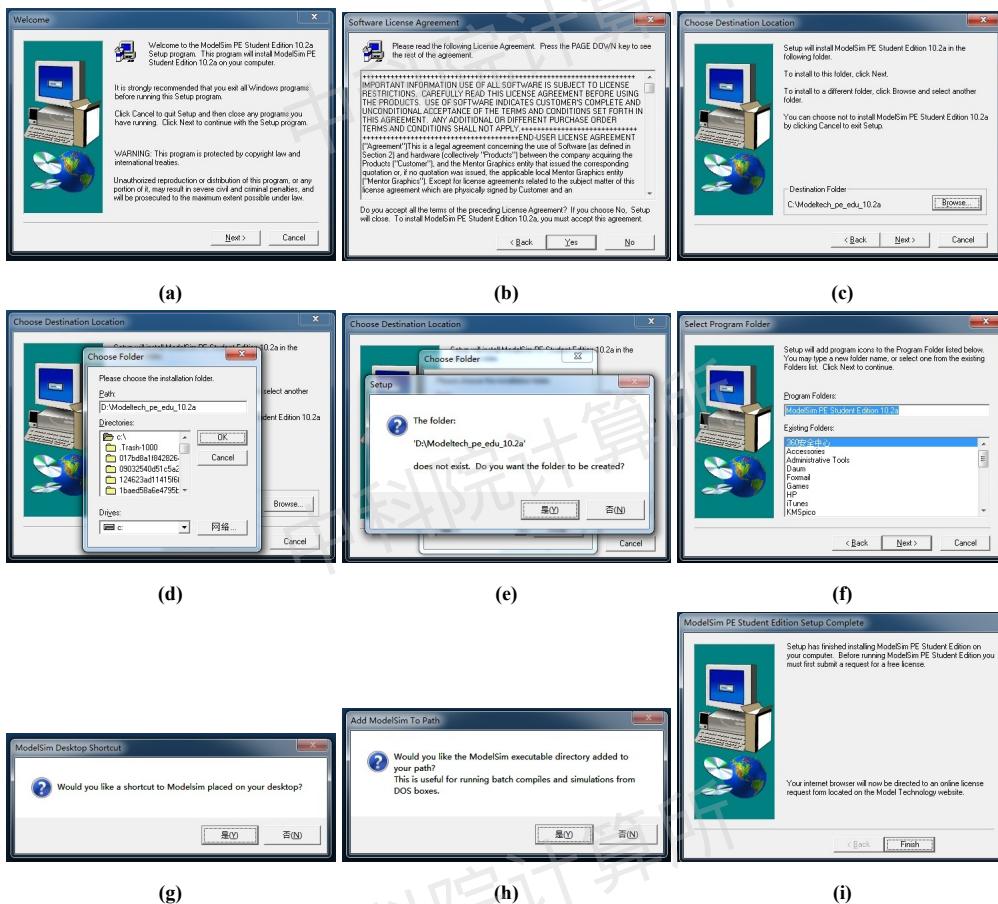


图 6.14 Modelsim 安装步骤

- (1) 运行安装包，进入如图6.14a所示初始界面，确认是否继续安装程序
- (2) 点击“Next”确认继续安装，进入如图6.14b所示 license 说明界面。
- (3) 点击“Yes”同意 license 说明内容，进入如图6.14c所示选择软件安装路径界面。
- (4) 点击“Browse”选择自定义安装路径，进入如图6.14d界面。
- (5) 在“Path”栏填写软件安装路径，并点击“OK”，进入图6.14e界面。
- (6) 第一次安装时，软件将在指定路径下创建 Modelsim 文件夹，点击“是”确认创建文件夹，进入图6.14f界面。

- (7) 点击“Next”确认继续安装，进入如图6.14g界面。
- (8) 点击“是”确认在桌面创建软件快捷方式，进入如图6.14h界面。
- (9) 点击“是”确认将软件安装路径加入系统环境变量中，便于在 DOS boxes 下运行批量处理脚本。安装程序进入如图6.14i界面。
- (10) 点击“Finish”后弹出申请 license 网页，按照提示设置 license 后，软件即安装完毕。

6.5.2 目录和代码文件

实验环境文件夹组织如下：

- 目录 src，包含编写的矩阵运算子单元 verilog 源代码。
- 目录 sim，包含仿真脚本文件和顶层文件。
 - 文件 tb_top.v，测试顶层文件，生成激励信号，例化矩阵运算子单元模块。
 - 文件 build.do，编译脚本文件，如图6.15所示。
 - 文件 compile.f，编译文件列表，如图6.16所示。
 - 文件 sim_run.do，仿真执行文件，如图6.17所示。关键参数含义为
 - * +nowarnTSCALE 表示忽略没有 timescale 定义的文件，用前面的 timescale 替代。
 - * -lib work 表示被仿真的库 (lib) 为 work。
 - * -c 表示从命令行启动仿真。
 - * -novopt 表示仿真时不要优化中间变量，保持最大的信号可观测性。
 - * tb_top 表示仿真顶层模块名为 tb_top。
- 目录 data，包含仿真输入输出数据文件。
 - 向量规模描述文件 scale。
 - 输入神经元文件 neuron。
 - 输入权值文件 weight。
 - 输出结果文件 result。

```

1 # 设置方针环境路径
2 # TODO
3 set sim_home Path/to/simulation/Dir
4
5 # 在当前目录下创建一个叫做work的目录，在里面存放方针数据文件
6 vlib ${sim_home}/work
7
8 #将work目录下的数据文件应设为一个叫做work的方针哭
9 vmap work ${sim_home}/work
10
11 #编译compile.f文件中指定的代码
12 vlog -f ${sim_home}/compile.f

```

图 6.15 编译脚本

tb_top.v 将读取控制信号文件数据输入矩阵运算子单元控制信号端口，并读取神经元数据和权值数据文件输入矩阵运算子单元的神经元端口和权值端口。Verilog 语法中，系统

```
1 // 源代码文件
2 path/to/src_dir/module_0.v
3 path/to/src_dir/module_1.v
4
5 // 顶层测试文件
6 path/to/sim_dir/tb_top.v
```

图 6.16 编译列表

```
1 vsim +nowarnTSCALE -lib work -c -novopt tb_top
```

图 6.17 执行脚本

任务 \$readmemb 和 \$readmemh 用来从文件中读取数据到存储器中。对于 \$readmemb 系统任务，每个数字必须使二进制数字，对于 \$readmemh 系统任务，每个数字必须使十六进制数字。这两个系统函数可以在仿真的任何时刻被执行使用，使用格式共六种：

- \$readmemb(“数据文件名”，存储器名);
- \$readmemb(“数据文件名”，存储器名，起始地址);
- \$readmemb(“数据文件名”，存储器名，起始地址，结束地址);
- \$readmemh(“数据文件名”，存储器名);
- \$readmemh(“数据文件名”，存储器名，起始地址);
- \$readmemh(“数据文件名”，存储器名，起始地址，结束地址);

其中，“数据文件名”指示被读取的数据文件，包含输入文件的路径和文件名，如图6.18所示。

```
1 initial
2 begin
3   $readmemb("D:/pe_exp/data/inst", inst);
4   $readmemh("D:/pe_exp/data/neuron", neuron);
5   $readmemh("D:/pe_exp/data/weight", weight);
6   $readmemh("D:/pe_exp/data/result", result);
7 end
```

图 6.18 设置仿真数据路径

6.5.3 编译调试

工具安装完毕后，可按照如下流程编译和调试代码。

6.5.3.1 编译代码

打开 ModelSim 软件，在命令行窗口输入“do path to build/build.do”命令，工具开始编译测试顶层文件和源代码。ModelSim 命令行窗口中将显示编译的文件、顶层文件已经编译错误和警告数量，如图6.19所示。

```
# |-- Compiling module tb_top
# -- Compiling module fifo
# -- Compiling module mcpu_mult
# -- Compiling module mcpu_acc
# -- Compiling module int45_to_fp_stg1
# -- Compiling module int45_to_fp_stg2
# -- Compiling module int45_to_int16
# -- Compiling module mcpu_cvt
# -- Compiling module mcpu
#
# Top level modules:
#   tb_top
# End time: 20:13:52 on Feb 14, 2020, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
```

图 6.19 编译日志

6.5.3.2 启动仿真

在 ModelSim 命令行窗口输入“do path to build/sim_run.do”命令，启动仿真。软件将显示如图6.20所示 sim instance 面板，以及仿真模块的层次关系和模块的信号名称。

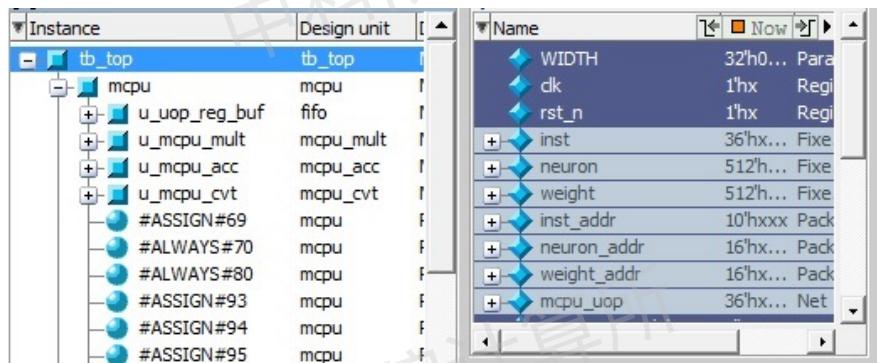


图 6.20 sim instance 面板

6.5.3.3 添加观测信号

在 sim instance 面板，先选择要调试的模块，然后在 Object 面板中，选择要观察的信号，单击右键“Add to -> Wave -> Selected Signals”将选中的信号添加到 Wave 窗口。

6.5.3.4 运行仿真

在命令行窗口输入“run all”命令，进行仿真，仿真完成后可通过波形窗口检查观察信号仿真的波形结果。

6.5.3.5 迭代调试

当发现仿真结果错误后，重新修改代码，执行以下步骤：

- 运行“do path to build/build.do”重新编译。
- 运行“restart”命令重新启动仿真。
- 运行“run -all”命令，执行仿真观察结果。
- 仿真完成后，使用“quit -sim”退出仿真。

6.6 实验评估

根据6.3节介绍完成矩阵运算子单元代码，然后通过 ModelSim 仿真 4 组不同规模的向量进行内积运算结果。`inst` 文件描述需进行内积计算的向量规模；`neuron` 文件存储输入的神经元数据；`weight` 文件存储输入的权值数据；`result` 文件存储运算结果。实验实现的串行内积运算器、并行内积运算器、矩阵运算子单元和 `result` 文件中结果都能比对正确 (`tb_top` 中 `compare_pass` 一直为 1)，说明实现的串行内积运算器、并行内积运算器和矩阵运算子单元功能正确。

本次实验的评估标准设定如下：

- 60 分：完成串行内积运算器，输出结果和 `result` 文件比对正确。
- 100 分：完成串行和并行内积运算器，输出结果和 `result` 文件比对正确。
- 120 分：完成串行内积运算器、并行内积运算器和矩阵运算子单元，输出结果和 `result` 文件比对正确。

6.7 实验思考

对比串行内积运算器和并行内积运算器，请说明深度学习处理器加速卷积计算的原理是什么？

第7章 综合实验

前述章节实验完成了图像处理领域中的风格迁移应用在智能处理器上的迁移，开发和优化。随着人工智能在更广泛的应用领域取得了良好效果，本章的综合实验将涉及在多个不同领域（如目标检测、文本检测、自然语言处理等）的人工智能应用在智能处理器上的开发和优化。

具体而言，第7.1节介绍基于 YOLOv3 网络模型^[24] 实现目标检测应用，并借助 TensorFlow 框架在智能处理器上进行部署。主要实验内容是完成非极大值抑制（Non-Maximal Suppression, NMS）的 BCL 实现并实现 YOLOv3 模型的 DLP 运行。

第7.2节介绍基于 EAST 网络模型^[25] 实现文字检测应用在智能处理器上的部署。主要实验内容为采用 BCL 实现 EAST 网络模型中的 Split+Sub+Concat 合并算子，并完成 EAST 网络在 DLP 上的部署。

第7.3介绍基于 BERT 网络模型^[26] 在智能处理器上的部署。主要实验内容为采用 BCL 实现 BERT 网络模型中的批量矩阵乘算子，并完成 BERT 网络在 DLP 上的部署。

7.1 目标检测-YOLOv3

7.1.1 实验目的

本实验主要完成将目标检测的代表性算法——YOLOv3 网络移植到深度处理器 DLP 上并进行性能优化，使读者可以借助 DLP 处理器完成完整的目标检测任务。

因此，本实验的主要目的包括：

1. 通过用智能编程语言实现 YOLOv3 的部分后处理操作，深入掌握智能编程语言的算子开发和优化方法；
2. 通过在 TensorFlow 中添加大算子，掌握在 TensorFlow 框架中添加融合算子的方法；
3. 通过使用 TensorFlow 进行在线推理，掌握使用 TensorFlow 编写目标检测应用并在典型 DLP 上进行优化的方法；

实验时间预计为两周。

7.1.2 背景介绍

7.1.2.1 目标检测

目标检测是指针对一个或多个特定类别在数字图像中进行视觉对象的分类和定位。目标检测属于多任务学习，也就是说一个任务分支需要通过分类算法将物体根据类别划分，另一个任务分支是在判断类别后标记出每一个类别的具体位置信息，即中心坐标位置或及坐标位置（以目标图片的左上角为原点 (0,0)）。随着技术的发展，目标检测逐渐成为其他计算机视觉任务的基础，如图像分割、图像字幕、物体追踪等；目前，目标检测已广泛应用于自动驾驶、机器人视觉、安防监控等领域。目标检测的两个任务分支如下图 7.2 所示：

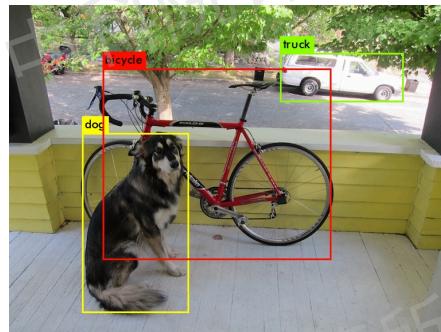


图 7.1 目标检测效果

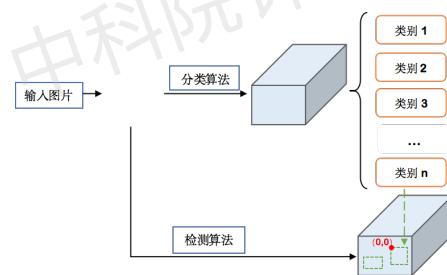


图 7.2 目标检测的多个任务分支

在介绍目标检测算法原理前，先列举一些目标检测中的常见概念如表 7.1 所示，包括平均精度均值（mean Average Precision, mAP）、滑动窗口（Sliding Window）、选择搜索算法（Selective Search, SS）、交并比（Intersection over Union, IoU）、兴趣区域（Region of Interest, ROI）、非极大值抑制（Non-Maximum Suppression, NMS）等。

表 7.1 目标检测常见知识

名称	定义
平均精度均值 mAP	AP 定义为 Precision-recall 曲线下的面积，mAP 为多个检测目标类别的 AP 均值，区间在 [0,1] 之间，越接近 1 证明精确度越高
滑动窗口 Sliding Window	采用不同大小和比例（宽高比）的窗口在全图片上以一定的步长进行滑动，以便于对应的区域做图像分类
选择搜索算法 Selective Search	将图片分为多个子区域，根据子区域的相似性不断迭代合并，在此过程中对合并区域实现外切矩形，得到候选框
交并比 IOU	用于预测 Bounding-box 和人工标注 Bounding-box 的重叠度，两个矩形框的 IOU 计算公式： $IOU = (A \cap B) / (A \cup B)$
兴趣区域（RoI）	从被处理的图像以方框或不规则形状等方式勾勒出需要处理的区域
非极大值抑制 NMS	依据候选区域和 score 矩阵，判断置信度最高的 bounding box，若预测框重叠，则选择置信得分高的边框

依据深度学习算法阶段不同，目标检测方法主要可以分为两类：一种是基于 Region Proposal 的两阶段检测，它首先由算法计算出目标候选框，然后再通过卷积神经网络对目标候选框进行分类与回归，主流的算法有 R-CNN、Fast R-CNN、Faster R-CNN 等；另一种是单阶段检测，不再需要候选区域，仅使用卷积神经网络直接将目标定位框转为回归问题，预测不同类别的目标位置，主流的算法有 YOLO、SSD 等。

7.1.2.2 YOLOv3

针对两阶段检测算法存在的计算冗余、检测速度较慢等问题，YOLO（You Only Look Once）算法提出“一步完成”的端到端检测器，将目标检测和定位统一到一个神经网络中。

本节实验使用经典的单阶段检测方法——YOLOv3 网络做目标检测。YOLOv3 延续了 YOLOv1 和 YOLOv2 的主要特点，包括划分单元格来做检测、使用 Leaky ReLU^[27] 作为激活函数、使用 Batch Normalization^[16] 避免过拟合、使用多尺度训练等。在此基础上，YOLOv3 通过修改主干网络来提升精度与性能。

YOLOv3 做目标检测时，首先用卷积神经网络提取特征，然后用非极大值抑制（NMS）来筛选候选框。本节实验需要采用智能编程语言来实现 NMS 算法。

- 网络结构

YOLOv3 的结构图如图 7.3 所示，输入图片经过 Darknet-53，然后通过 Concat 拼接输出 3 个不同尺度的特征图。

其中 DBL 是 Yolov3 的基本计算单元，它由一个 DarknetConv2d、一个 BatchNorm 和一个 LeakyRelu 操作线性排列组成，BatchNorm 和 LeakyRelu 是 conv2d 卷积层后必不可少的部分（不包括最后一个卷积层），每个算子的功能如下表 7.2 所示：

表 7.2 DBL 的基本计算单元

算子名称	功能
DarknetConv2d	Darknet 是 Yolov3 的基础网络，1 个 Darknet 的 2 维卷积 Conv2D 层，即 DarknetConv2D(); 操作流程：通过 kernel_regularizer 将卷积核的参数进行 L2 正则化，当 Padding 的 strides= (2,2) 时采用 valid 模式，其他 Padding 一般采用 same 模式；其余操作与常规二维卷积 Conv2D() 保持一致。
BN	BatchNormalization(), 将输入数据进行正则化，把每层神经网络的输入值的分布强制拉回到均值为 0 方差为 1，的标准正态分布，得到的输出值作为激活输入数据，分布在非线性函数的敏感区，加快收敛速度。
Leaky()	LeakyReLU 激活函数是 ReLU 的变换，公式为： $y = \max(0, x) + leak * \min(0, x)$; (斜率为 0.1)

Res_unit 是 Resblock_body 的基本组件之一，它是由两个 DBL 和一个 ADD 组合，借鉴了 Resnet 的残差结构，即将每个 Res_unit 的输入经过两次 DBL 处理后生成的特征图与输入自身进行叠加（Add），Add 操作仅是直接相加并不会导致 Tensor 维度的改变。引入残差结构使 Yolov 网络深度从 Darknet-19 (Yolov2) 增加至 Darknet-53 (Yolov3)，残差层可以减小网络过深而引起的梯度爆炸风险，提高网络的训练效率。

Resblock_body 在网络结构中作为 Yolov3 的大组件，用 Res_n 表示，n 为在此结构中 Res_unit 的个数，它是由一个 ZeroPadding2D、一个 DBL、N 个 Res_unit 线性排列组成。ZeroPadding2D 根据下一步操作的需要，将输入矩阵的边界进行 0 充填，以达到信息补齐的效果。

Concat 是将不同层的 Tensor 进行拼接，在 Yolov3 中主要是将 darknet 中间层的输出与上采样结果进行维度扩张，以获得不同的输出。

Darknet.output 输出的底层信息包含全局特征信息，DarkNet 中间层包含局部特征信息，通过 Concat 拼接，可提高检测精度。

值得注意的是 Yolov3 的对象分类预测并没有延续 Yolov2 中的 Softmax，而是采用了 Logistic Regression 对输出的 bbox 进行预测类别，这是因为 Softmax 要求每个目标对象具有相互独立的标签，而数据集中每个对象需要支持多个标签，将 Softmax 用 Logistic 可以预测每个类别的得分。

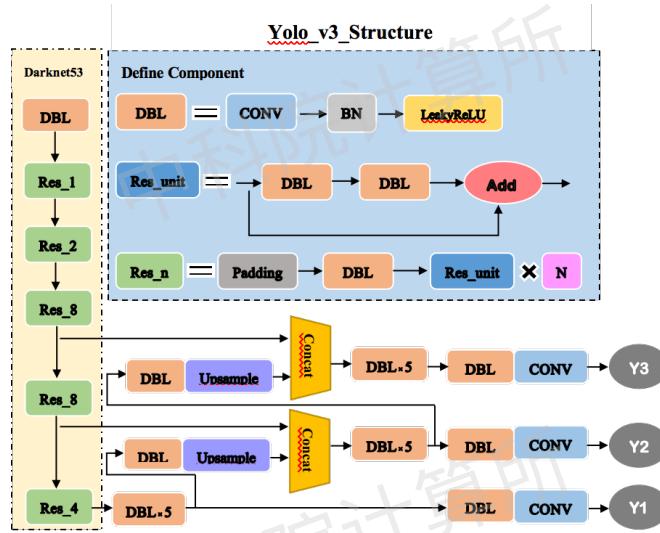


图 7.3 YOLOv3 结构图

对于网络输出，Yolov3 中的获得 anchorbox 的方案依旧使用了 Yolov2 的聚类法，首先计算出 9 个聚类中心，再按照大小均匀分给 3 个尺度，经过卷积层对输出采用多尺度检测技术，也就是说有个三个不同尺度的输出特征图，分别为 13×13 、 26×26 、 52×52 ，其深度都为 255。由于每个网络单元都有 3 个检测框，每个框包含检测框中心坐标 (x,y)、检测框的长度 (h)、检测框的宽度 (w) 以及检测框的置信度 (confidence) 五个基本参数，80 个检测类别都有对应的概率，所以特征图的深度为 255 ($3 \times (5 + 80) = 255$)。三个不同尺度的输出如下表 7.3 所示，对应与图 7.3 中的三个输出 (Y1、Y2 和 Y3)：

表 7.3 YoloV3 的网络输出

尺度	检测类型	实现方式
13×13	大型目标	Darknet 的网络最终输出结构为 $13 \times 13 \times 1024$ ，通过设置 512 通道个数和 $num_anchors * (num_classes + 5)$ ，得到预测矩阵 Y1。
26×26	中型目标	通过 DBL 将通道数由 512 调整至 256，再经过上采样 UpSampling2D，将 13×13 的网络结构转换为 26×26 ，得到的网络结构图与 DarkNet 的中间层输出结果进行拼接，最终结果即是第 2 个尺度特征图 Y2。
52×52	小型目标	通过 DBL 将通道数由 256 调整至 128，再经过上采样 UpSampling2D，将 26×26 的网络结构转换为 52×52 ，得到的网络结构图与 DarkNet 的中间层输出结果进行拼接，最终结果即是第 3 个尺度特征图 Y3。

YOLOv3 中使用 Darknet-53 作为 backbone 网络用以提取图片的特征信息，相较于 YOLOv2 中的 Darknet-19，其使用步长为 2 的卷积来替代之前的池化操作，同时采用 ResNet^[2] 的残

差结构 (resblock) 提升精度。与 ResNet-152 和 ResNet-101 相比, Darknet-53 网络层数较少, 不仅分类精度没有下降, 检测速度还有所提升。图 7.4 展示了 Darknet-53 的结构, 通过 5 次步长为 2 的卷积进行尺寸变换, 最终输出的 Feature Map 为输入的 1/32。

Type	Filters	Size	Output
Convolutional	32	3×3	256×256
Convolutional	64	$3 \times 3 / 2$	128×128
1x	32	1×1	
Convolutional	64	3×3	128×128
Residual			
Convolutional	128	$3 \times 3 / 2$	64×64
Convolutional	64	1×1	
2x	128	3×3	
Residual			64×64
Convolutional	256	$3 \times 3 / 2$	32×32
Convolutional	128	1×1	
8x	256	3×3	
Residual			32×32
Convolutional	512	$3 \times 3 / 2$	16×16
Convolutional	256	1×1	
8x	512	3×3	
Residual			16×16
Convolutional	1024	$3 \times 3 / 2$	8×8
Convolutional	512	1×1	
4x	1024	3×3	
Residual			8×8
Avgpool		Global	
Connected		1000	
Softmax			

图 7.4 Darknet-53 结构图

Yolov3 中的 Darknet-53 网络结构有以下两个特点:

1. 将全卷积层应用于整个网路: Darknet-53 是 53 个卷积层和池化层的组合,但在 Yolov3 中并没有包含 darknet-53 最后一个全局平均池化层,所以减小张量维度是通过改变 5 次卷积核的步长来实现的。全卷积结构也应用于预测分支,最后一个卷积核的规模为 $1 \times 1 \times 1024 \times 255$, 其中 255 是针对 COCO 数据集的 80 类别确定的卷积核个数。
2. 借鉴 Resnet 思想,引入残差 (Rsidual) 结构: 网络深度越深, 特征信息表达越好, 但随着网路深度的加深, 训练误差趋势会先下降再上升, 也就意味着网络深度越深, 模型训练难度越大; 所以采用残差模块不仅可以保证在网络深度加深的情况下仍能保持收敛状态, 还在一定程度上大大减少了计算量。

• 非极大值抑制: NMS

得到 3 种不同尺度的 Feature Map 后需要进行后处理操作来筛选出合适的框以完成检测, NMS 非极大值抑制是其中重要的操作。NMS^[28] (非极大值抑制) 是一种针对目标框的多重性而提出的检测技术, 是目标检测网络中一个重要环节, 由于目标检测相邻的交叉窗口经常出现分数相近的情况, 使用 NMS 可以去掉重复的检测框。下图 7.5 展示了从输出预选框到生成最终检测框的过程。

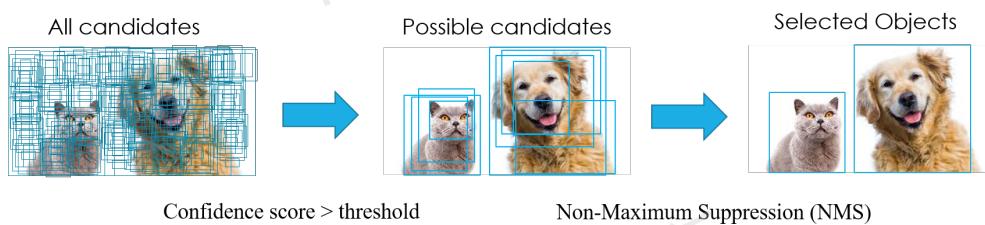


图 7.5 NMS 示意图

NMS 的计算过程可概括为以下四个步骤:

1. 对所有预测框的置信度降序排序；
2. 选出置信度最高的预测框，确认其为正确预测，并计算它与其他预测框的 IOU；
3. 根据（2）中计算的 IOU 去除重叠度高的框，即 $\text{IOU} > \text{thresh_iou}$ 就删除；
4. 剩下的预测框返回第 1 步，直到没有置信度 $> \text{thresh_score}$ 的框为止。

其中，交并比 IOU 的计算可以参考《智能计算系统》教材^[1] 第 3.3.1.1 节。

由于目标检测类算法一般都包含 NMS 计算环节，之外许多其它场景下也可以用到 NMS 计算过程。在这样的背景下，有必要将 NMS 模块在 DLP 硬件上的实现整体抽象剥离出来，单独封装成一个 NMS 通用模块，不仅适用于任何检测算法，还可以缩短其它检测算法的开发时间，增加代码的可维护性与稳定性。因此检测网络综合实验中设计了一个 NMS 通用模块任务，以.h 文件提供给调用程序，将 NMS 计算过程封装成 nms_detection 函数，供 YOLOv3 后处理直接调用。

7.1.3 实验环境

本节实验所涉及的硬件平台和软件环境如下：

- 硬件平台：硬件平台基于前述的 DLP 云平台环境。
- 软件环境：所涉及的 DLP 软件开发模块包括编程框架 TensorFlow、高性能库 CNML、运行时库 CNRT、编程语言及编译器、运行时库 CNRT。

7.1.4 实验内容

本节实验主要是在 DLP 硬件上实现并优化以 YOLOv3 网络模型为核心的目标检测应用。针对用户从网络上下载的标准 YOLOv3 模型，经过模型量化并配置 DLP 相关参数，则可以采用 DLP 上的定制 TensorFlow 版本来运行。为了充分发挥 DLP 的计算能力，进一步采用智能编程语言 BCL 实现 YOLOv3 中的 NMS 部分，并将其作为一个大算子集成到 TensorFlow 框架中。考虑推理应用的实际需求，采用 DLP 的流式框架编写完整目标检测应用，基于离线模型进行部署。具体实验内容包括：

1. **YOLOv3 DLP 运行**：将标准的 YOLOv3 网络模型进行量化后，通过定制的 TensorFlow 版本在 DLP 硬件上运行；
2. **NMS 的 BCL 实现**：采用 BCL 实现 NMS 后处理运算；
3. **框架算子集成**：将用 BCL 实现的 NMS 算子集成到 TensorFlow 框架中，进行在线推理。

图7.6 虚线框中的是本实验需要修改的代码文件。在 BCL 实现部分主要包括了 nms_detection 代码的编写和 CNPlugin 的集成。在 CNPlugin 集成完成之后需要将算子集成到 TensorFlow 中。最后在完成所有框架修改工作后执行模型的现在推理。

7.1.5 实验步骤

如前所述，详细的实验步骤主要包括：YOLOv3 DLP 运行、NMS 的 BCL 实现、框架算子集成等。

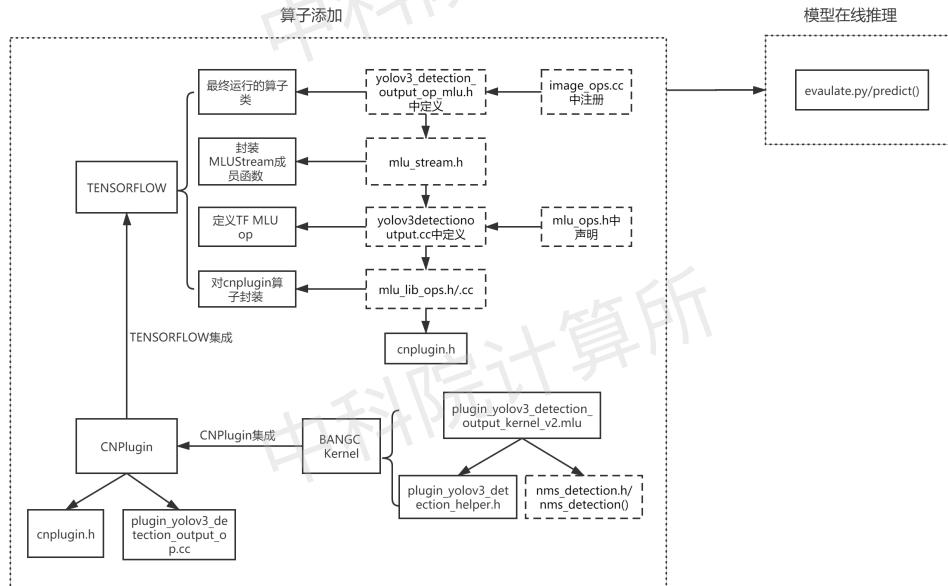


图 7.6 实验内容

7.1.5.1 YOLOv3 DLP 运行

为了使得标准的 YOLOv3 网络模型在 DLP 上运行，主要需要完成：YOLOv3 CPU 运行、模型量化和 DLP 运行等流程。

• YOLOv3 CPU 运行

通过在 CPU 上正确执行 YOLOv3 标准网络模型完成推理，确保相关依赖都已安装完成。具体步骤如下：

1. 获取开源 YOLOv3 工程：git clone <https://github.com/YunYang1994/tensorflow-yolov3.git>

2. 安装推理所需软件包：一些必备的 Python 安装包在所提供的云平台软件环境中已经安装完成，但对于每个综合实验的特有依赖还需进行单独安装，使用图 7.7 中的命令完成依赖的安装与升级。

```

1 $ cd tensorflow-yolov3
2 $ pip install -r ./docs/requirements.txt

```

图 7.7 YOLOv3 依赖安装

3. 模型下载与转换：执行图 7.8 中的命令完成模型的下载与转换。要执行这一步的原因是模型文件和 ckpt 文件之间有一些版本上的差异，需要将节点名等细节对齐。

4. 模型推理与验证：执行图 7.9 中的命令完成在 CPU 上的推理。

• 模型量化

下面介绍如何将开源的 pb 模型量化为 INT8 pb 模型。DLP 软件环境提供 fppb_to_intpb 工具来进行模型量化。该工具在 TensorFlow 根目录的 tensorflow/cambricon_examples/tools/fppb_to_intpb 目录下。具体的量化分为以下几个步骤：

```

1 $ cd checkpoint
2 $ wget https://github.com/YunYang1994/tensorflow-yolov3/releases/download/v1.0/
     yolov3_coco.tar.gz
3 $ tar -xvf yolov3_coco.tar.gz
4 $ cd ..
5 $ python convert_weight.py
6 $ python freeze_graph.py

```

图 7.8 YOLOv3 模型下载与转换

```

1 $ python image_demo.py

```

图 7.9 YOLOv3 CPU 推理

1. 生成数据文件:按照模型所需要的数据集以及数据集的具体路径来修改 generate_image_list.py 文件中的 image_libs_map，来生成相应的数据文件。YOLOv3 需要 coco 数据集，因此将 image_libs_map 进行如图 7.10 所示的修改。完成修改后执行 generate_image_list.py 来生成数据文件。

```

1 // filename: generate_image_list.py
2 import os
3 from os.path import isfile, join
4
5 max_image_count = 100
6 image_libs_map = {
7     "coco": join(os.environ.get("COCO_DATASET_HOME"), "COCO/test2017")
8 }
9 if __name__ == "__main__":
10     for image_lib, image_path in image_libs_map.items():
11         n = 0
12         with open("image_list_{}".format(image_lib), "w") as image_list:
13             for root, dirs, files in os.walk(image_path):
14                 for f in files:
15                     if n >= max_image_count:
16                         break
17                     if f.endswith("jpg"):
18                         image_list.write(join(image_path, f) + "\n")
19                     n = n + 1

```

图 7.10

2. 生成量化配置文件:在进行量化前需设置相应的配置文件。以指定量化相关的参数。DLP 软件环境提供了 generate_ini.py 来生成指定模型的量化配置文件。针对 YOLOv3，需要将其中的 model_name 进行如图 7.11 所示的修改，即指定要生成的配置文件是针对 YOLOv3 的。

```

1 models_name = ['yolov3']

```

图 7.11

执行 generate_ini.py 脚本后会生成 config 文件夹，其中包含了 yolov3_naive_int8.ini 参数配置文件。该配置文件中包含如图 7.12 所示的信息。

```

1 // filename: yolov3_naive_int8.ini
2 [preprocess]
3 mean = 0, 0, 0          #均值，顺序依次为 mean_r、mean_g、mean_b
4 std = 255.0              #方差
5 color_mode = rgb         #网络的输入图片是rgb还是bgr
6 crop = 416, 416          #前处理最终将图片处理为 416 * 416 大小
7 calibration = yolov3_preprocess_cali    #校准数据读取及前处理的方式，可以根据需求进行自
     定义，[preprocess] 和 [data] 中定义的参数均为 calibration 的输入参数
8
9 [config]
10 activation_quantization_alg = naive #输入量化模式，可选 naive 和 threshold_search，naive
      为基础模式，threshold_search 为阈值搜索模式
11 device_mode = clean               #可选 clean、mlu 和 origin，建议使用 clean，使用 clean
      生成的模型在运行时会自动选择可运行的设备
12 use_convfirst = False            #是否使用 convfirst
13 quantization_type = int8        #量化位宽，目前可选 int8 和 int16
14 debug = False                  #是否为 debug 模式
15 weight_quantization_alg = naive #权值量化模式，可选 naive 和 threshold_search，naive 为
      基础模式，threshold_search 为阈值搜索模式
16 int_op_list = Conv, FC, LRN    #要量化的 layer 的类型，目前只能量化 Conv、FC 和 LRN
17 channel_quantization = False   #是否使用分通道量化，目前不支持为 True
18
19 [model]
20 output_tensor_names = pred_sbbox(concat_2:0, pred_mbbox(concat_2:0, pred_lbbox(concat_2
      :0))           #输出 Tensor 的名字，可以是多个，以逗号隔开
21 original_models_path = ./realpath-of-yolov3_coco.pb #输入pb
22 save_model_path = ./pbs/yolov3/yolov3_int8.pb       #输出pb
23 input_tensor_names = input/input_data:0             #输入 Tensor 的名字，可以是多个，以逗号隔开
24
25 [data]
26 num_runs = 2          #运行次数
27 data_path = ./image_list_coco #校准数据文件路径
28 batch_size = 10 #每次运行的batch_size

```

图 7.12

其中，可以通过 activation_quantization_alg 和 weight_quantization_alg 来设置具体的量化模式。包括 naive 和 threshold_search 两种方式。

native 模式对应于常规量化方式，直接统计数据的最大值和最小值完成量化。

threshold_search 阈值搜索模式用于处理存在异常值的待量化数据集，该模式能够过滤部分异常值，重新计算出数据集的最值，用新最值来计算数据集的量化参数，从而提高数据集整体的量化质量。在这种模式下，数据量化所使用的最大值和最小值不再由简单的统计得出，而是通过搜索的方式得到。但是对于不存在异常值，数据分布紧凑的情况下，不建议使用该算法，比如网络权值的量化。

其中 device_mode 可以设置输出 pb 所有节点的 device，有三种配置方式：clean、mlu 和 origin。其中，mlu 将输出 pb 的所有节点的 device 都设置为 MLU，即都在 MLU 上运行；clean 将输出 pb 的所有节点的 device 清除，运行时根据算子注册情况自动选择可运行的设备；origin 则使用和输入 pb 一样的设备指定。

3. 完成模型量化：运行量化工具 python fppb_to_intpb.py config/yolov3_naive_int8.ini 完成模型的量化。如7.12所示。

- DLP 运行

完成 CPU 侧的推理后，接下来需要将其移植在 DLP 上，以加快其推理速度。此时我们只需仿照编程语言算子实验中的 DLP 推理移植部分，通过配置 session config 以及进行 INT8 量化即可完成 DLP 移植。下面将详细介绍这两个步骤：

修改上述执行 CPU 推理时的代码 `image_demo.py`，增加如 7.13 的几个部分。

```

1 // filename: image_demo.py
2 pb_file = "./realpath-to-int8-pb"
3 ...
4 config = tf.ConfigProto(allow_soft_placement=True,
5                         inter_op_parallelism_threads=1,
6                         intra_op_parallelism_threads=1)
7 config.mlu_options.data_parallelism = 1
8 config.mlu_options.model_parallelism = 1
9 config.mlu_options.core_num = 4
10 config.mlu_options.core_version = "MLU270"
11 config.mlu_options.precision = "int8"
12 config.mlu_options.save_offline_model = False
13 config.mlu_options.offline_model_name = "yolov3_int8.cambricon"
14 with tf.Session(config=config, graph=graph) as sess:
15     ...

```

图 7.13 YOLOv3 DLP 移植（1）

7.1.5.2 NMS BCL 代码实现

NMS 模块代码为 `nms_detection.h`。在使用 BCL 设计 NMS 模块时，需要考虑以下设计原则：

1. 对于功能模块可能面对的不同规模数据，能够接受的输入数据来源为 GDRAM, NRAM, SRAM 中的一种；
2. 同样对于不同规模的数据，能够将输出数据存放到 GDRAM, NRAM, SRAM 中的一种；
3. 能够以不同的储存格式存放数据；
4. 能够在 BLOCK 和 U1 的模式下工作；
5. 能够处理尽量多样的数据规模；

使用 BCL 实现时，如果按照背景介绍章节中 NMS 原理进行计算，则会显得较为复杂，比如第(3)步中删除交并比大于阈值的框，使用 BCL 语言实现这个操作需要来回拷贝数据，导致效率不高。为了解决此问题，标准程序中采用另外一种方式，即不改变之前的数据，将被删除框对应的分数置为 0，下一轮筛查的时，删除的框必然不会对此轮筛查造成影响。整体设计流程图如图 7.14 所示：

下面以图 7.15 中的代码为例进行说明。NMS 通用模块接口介绍：

- `output_box_num`: NMS 筛选出的框的总个数；
- `output_data`: NMS 计算结果存放的数据地址；
- `dst`: NMS 计算结果存放的数据地址类型：GDRAM/SRAM/NRAM；
- `input_data_score`: 输入框 score 的数据地址；

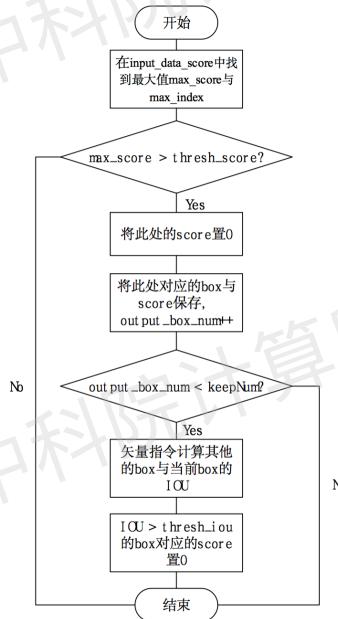


图 7.14 BangC NMS 设计流程图

```

1 // filename: nms_detection.h
2 __mlu_func__ void nms_detection( int& output_box_num ,
3                                 NMS_DT* output_data ,
4                                 Addr dst ,
5                                 NMS_DT* input_data_score ,
6                                 NMS_DT* input_data_box ,
7                                 Addr src ,
8                                 NMS_DT *buffer ,
9                                 int buffer_size ,
10                                NMS_DT* sram ,
11                                SplitMode split_mode ,
12                                int input_box_num ,
13                                int input_stride ,
14                                int output_stride ,
15                                int keepNum ,
16                                NMS_DT thresh_iou ,
17                                NMS_DT thresh_score ,
18                                int save_method );
19
20

```

图 7.15 接口设计

- **input_data_box**: 输入框坐标的数据地址, 储存顺序是: x1, y1, x2, y2, 同一个类型的数据储存在一起;
- **src**: NMS 输入数据存放的数据地址类型: GDRAM/SRAM/NRAM;
- **buffer**: NMS 计算使用的 NRAM 空间首地址;
- **buffer_size**: NMS 计算使用的 NRAM 空间大小, 单位为字节, 当 dst 为 GDRAM/S-RAM, 则 buffer_size 至少需要: $(64 * 4 + 64 + 256 * 5) * \text{sizeof}(\text{NMS_DT})$ 字节; 当 dst 为 NRAM, 则 buffer_size 至少需要: $(64 * 4 + 64) * \text{sizeof}(\text{NMS_DT})$ 字节。注意: 如果 src ==

NRAM 并且 input_box_num 是对齐的, NMS 计算会更加高效, 因为可以省去数据搬运工作。

• sram: SRAM 的地址空间, 用来在 U1 模式下通过核间通讯找 score 的最大值, 大小至少为 $30 * \text{sizeof}(\text{NMS_DT})$ 字节;

- split_mode: 拆分模式: NMS_BLOCK/NMS_U1;
- input_box_num: 输入框的数量;
- input_stride: 输入数据的 stride, 即 $x1, y1, x2, y2$ 之间的 stride;
- output_stride: 输出数据的 stride, 即 $x1, y1, x2, y2$ 之间的 stride;
- keepNum: 最多保留筛选框的个数;
- thresh_iou: 交并比阈值;
- thresh_score: score 阈值;
- save_method: 储存模式, 分为 0/1/2

具体实现主要分为两个步骤:

1. 准备阶段: 准备所需的变量, 防呆检测, 空间划分, 多核拆分

(a) 主要变量有:

• core_limit: 启用的核数, BLOCK 模式对应 1, U1 模式对应 4;

• loop_end_flag: U1 模式结束标识;

• nram_save_limit_count: NRAM 上临时储存筛选框的数量;

• MODE: 当 $\text{src}=\text{NRAM}$ 时, 0 代表数据需要先加载到指定的 NRAM 上进行计算, 1 代表数据直接在 NRAM 上可以直接运算, 保证高效, 但需要满足两个条件: buffer 空间足够, input_box_num 是对齐的;

- input_score_ptr: 输入框 score 的数据指针;
- input_x1_ptr: 输入框 box 的 $x1$ 坐标数据指针;
- input_y1_ptr: 输入框 box 的 $y1$ 坐标数据指针;
- input_x2_ptr: 输入框 box 的 $x2$ 坐标数据指针;
- input_y2_ptr: 输入框 box 的 $y2$ 坐标数据指针;
- $x1$: buffer 空间, 存放 $x1$;
- $y1$: buffer 空间, 存放 $y1$;
- $x2$: buffer 空间, 存放 $x2$;
- $y2$: buffer 空间, 存放 $y2$;
- score: buffer 空间, 存放 score;
- inter_x1: buffer 空间, IOU 筛选临时空间;
- inter_y1: buffer 空间, IOU 筛选临时空间;
- inter_x2: buffer 空间, IOU 筛选临时空间;
- inter_y2: buffer 空间, IOU 筛选临时空间;
- max_box: buffer 空间, 筛选框的信息储存空间, 顺序为 max score, $x1, y1, x2, y2$;
- nram_save: buffer 空间, 筛选框的临时储存空间;
- limit = 0: 根据 buffer 进行 findlimit, 一次最多能处理的输入框的个数;
- len_core = 0: 每个核处理的框的个数;
- max_seg_pad = 0: 每次处理输入框的个数, 根据 limit 进行下补齐, 满足硬件限制;

- repeat：整数段，需要处理几次 max_seg_pad；
- remain：余数段，剩下得需要处理的框的个数；
- remain_pad：余数段进行补齐后的框个数，满足硬件限制；
- input_offset：当前核处理的输入数据的起始地址偏移；
- nram_save_count：临时储存空间储存多少个框后再整体拷贝到实际的目标地址；

(b) 空间划分 MODE 为 0 的时候， $x_1, y_2, x_2, y_2, score$ 指向的 buffer 空间用来加载数据，MODE 为 1 的时候，不再需要加载数据的空间，inter_x1, inter_y1, inter_x2, inter_y2 是 buffer 上的临时空间，用来计算 NMS，4 块临时空间是考虑了空间复用后最少需要的数量，以节省 buffer，可以一次处理更多的数据。

(c) 多核拆分 src==NRAM 模块内只支持单核模式，用户可根据需求在模块外按类拆分，即每个核负责计算一个类的 NMS 过程，因为模块内无法在 U1 模式时对某一个核上的 NRAM 上的数据进行访问；src==GDRAM/SRAM，模块内支持按数据块进行多核拆分，即一个类的 NMS 过程拆分到多个核上进行计算。多核拆分使用的拆分方式是：每个核上分到框的数量之间相差不超过 1，即尽可能的将数据平均分配到多个核上。

2. 执行阶段：主要操作都放在 for (int keep = 0; keep < keepNum; keep++) 循环中，一次找一个框，这个循环有两个退出条件：找到的框数量达到 keepNum，最大的 score 小于 thresh_score。

(a) 搜索最大值模块：通过 _bang_max 指令找到 score 存放数据的最大值与最大索引，如果是 BLOCK 模式，则直接即可找到 score 的最大值与最大索引，如果是 U1 模式，则需要每个核上计算所分到数据的最大值，然后通过 SRAM 进行通信，进一步找到四个核上的最大值与最大值索引。

(b) 保存模块：保存方式主要包括两种，一种是按 score, x1, y1, x2, y2 的顺序一组一组进行储存，一种是按 score, x1, y1, x2, y2 的顺序以此存放。两者区别如下图 7.16 所示：

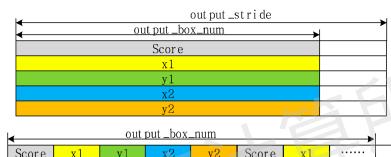


图 7.16 BangC NMS 两种保存方式

(c) 根据交并比筛选模块：以图 7.17 中的代码为例进行说明。

7.1.5.3 TensorFlow 算子添加

TensorFlow 中添加算子可分为两种情况，一种为 TensorFlow 官方定义的标准流程，可以从官方文档中获取；另一种为添加 MLU 算子的方式，本节将对如何添加 MLU YOLOv3 后处理算子作出介绍。

添加 MLU 算子可分为以下几个步骤：

1.MLUlib 层，该部分代码在 tensorflow/stream_executor/mlu/mlu_api/lib_ops/ 目录下，主

```

1 *IOU计算部分*
2 // 得到相交部分的面积:
3 __nramset(inter_y1, seg_len, max_box[1]); // max_x1
4 __svmax_relu(inter_x1, x1, inter_y1, seg_len); // inter_x1
5 __nramset(inter_y2, seg_len, max_box[3]); // max_x2
6 __svmin_relu(inter_x2, x2, inter_y2, seg_len); // inter_x2
7 __bang_sub(inter_x1, inter_x2, inter_x1, seg_len);
8 __bang_active_relu(inter_x1, inter_x1, seg_len); // inter_w
9 __nramset(inter_x2, seg_len, max_box[2]); // max_y1
10 __svmax_relu(inter_y1, y1, inter_x2, seg_len); // inter_y1
11 __nramset(inter_x2, seg_len, max_box[4]); // max_y2
12 __svmin_relu(inter_y2, y2, inter_x2, seg_len); // inter_y2
13 __bang_sub(inter_y1, inter_y2, inter_y1, seg_len);
14 __bang_active_relu(inter_y1, inter_y1, seg_len); // inter_h
15 __bang_mul(inter_x1, inter_x1, inter_y1, seg_len); // area_I
16 // 得到每个框的面积: area = (x2 - x1) * (y2 - y1):
17 __bang_sub(inter_y1, x2, x1, seg_len);
18 __bang_sub(inter_y2, y2, y1, seg_len);
19 __bang_mul(inter_x2, inter_y1, inter_y2, seg_len); // area
20 // 得到相并部分的面积area_U: area + max_area - area_I:
21 __nramset(inter_y1, seg_len, max_area);
22 __bang_add(inter_x2, inter_x2, inter_y1, seg_len);
23 __bang_sub(inter_x2, inter_x2, inter_x1, seg_len); // area_U
24 *筛选*
25 // 如果IOU超过阈值, 则将相应的框的score置0: area_U * thresh > area_I?
26 __bang_mul_const(inter_x2, inter_x2, thresh_iou, seg_len);
27 __bang_gt(inter_x1, inter_x2, inter_x1, seg_len);
28 __bang_mul(score, score, inter_x1, seg_len);

```

图 7.17 NMS 交并比筛选模块

要作用是进一步对 SDK 层进行 API 封装。需要注意的是 MLU LIB 层禁止引用 TensorFlow 的头文件，目前只使用了 tensorflow::Status 类方便做返回值处理。对于该算子来说主要对该目录下的 mlu_lib_ops.cc 和 mlu_lib_ops.h 文件进行修改，示例代码如下所示。

2.MLUOps 层实现，MLUOps 层的职责是使用 LIB 层的 API 完成算子的实现。该部分代码在 tensorflow/stream_executor/mlu/mlu_api/ops 下，注意每个算子首先需要在该目录下的 mlu_ops.h 文件中添加算子类的声明。下图 7.19 为添加算子类声明的程序。

添加完算子声明后还需同一个目录.cc 文件中添加算子的实现，每个算子都需实现 Create 和 Compute 两个方法。下图 7.20 和 7.21 程序显示了 yolov3detectionoutput.cc 的实现。

Create 方法的职责是将创建好的 baseop 指针调用 base_ops_.push_back(op_ptr); 储存起来；Compute 方法的职责在于调用 lib 层的 compute 函数进行计算，目前算子实现为同步方式，因此需要调用 SyncQueue。

3.MLUStream 实现，MLUStream 负责 MLUOps 算子类的实例化，其接口都定义在 tensorflow/stream_executor/mlu/mlu_stream.h 中。以 MLUOpKernel 和 MLUStream 层的接口特点，可以把 MLUStream 层的算子分为 2 类：通用模版算子与特例化算子。对于 Yolov3DetectionOutput 算子来说，其属于通用模版算子。

通用模版算子满足以下三个条件：MLU 算子的所有输入均来自 OpKernelContext；MLU 算子的所有输出顺序须与 OpkernelContext 的输出顺序一致；MLUTensor 可以被 CreateMLUTensorFromTensor 创建，即 MLU Tensor 的形状、数据类型与 TensorFlow tensor 一致。

```

1 // filename: mlu_lib_ops.cc
2 tensorflow::Status CreateYolov3DetectionOutputOp(
3     MLUBaseOp** op, MLUTensor** input_tensors, MLUTensor** output_tensors,
4     cnmlPluginYolov3DetectionOutputOpParam_t param) {
5     CNML_RETURN_STATUS(cnmlCreatePluginYolov3DetectionOutputOp(
6         op, param, input_tensors, output_tensors));
7 }
8
9 tensorflow::Status ComputeYolov3DetectionOutputOp(MLUBaseOp* op,
10                                                 MLUCnrtQueue* queue,
11                                                 void* inputs[], int input_num,
12                                                 void* outputs[],
13                                                 int output_num) {
14
15     int dp = 1;
16     cnrtInvokeFuncParam_t compute_forw_param;
17     u32_t affinity = 0x01;
18     compute_forw_param.data_parallelism = &dp;
19     compute_forw_param.affinity = &affinity;
20     compute_forw_param.end = CNRT_PARAM_END;
21
22     cnmlComputePluginYolov3DetectionOutputOpForward(
23         op, inputs, input_num, outputs, output_num, &compute_forw_param, queue);
24 }

```

(a)

```

1 // filename: mlu_lib_ops.h
2 tensorflow::Status CreateYolov3DetectionOutputOp(
3     MLUBaseOp** op, MLUTensor** input_tensors, MLUTensor** output_tensors,
4     cnmlPluginYolov3DetectionOutputOpParam_t param);
5
6 tensorflow::Status ComputeYolov3DetectionOutputOp(MLUBaseOp* op,
7                                                 MLUCnrtQueue* queue,
8                                                 void* inputs[], int input_num,
9                                                 void* outputs[],
10                                                int output_num);
11

```

(b)

图 7.18 MLULib 层实现

4. MLUOpKernel 实现, OpKernel 为 TensorFlow 对算子对抽象定义。MLUOpKernel 继承了 OpKernel 类, 其使用方法与 OpKernel 基本一致。对于每个 MLU 算子均需要实现其构造函数与 ComputeOnMLU 方法。MLUOpKernel 实现的主要功能有参数检查、参数处理、输出形状推断及输出内存分配、调用 MLUStream 层接口完成算子计算。对应代码为 yolov3_detection_output_op_mlu.h。下图 7.23、7.24、7.25 中的代码分别显示了这四个不同的步骤。input0、input1、input2 为 YOLOv3 结构图 7.3 中的三个 feature map 输出。

5. 开始注册 MLUOpKernel, 在 tensorflow/core/kernels 中添加 yolov3_detection_output_op.cc 注册文件和添加 yolov3_detection_output_op_mlu.h 算子实现文件, Op Kernel 注册使用的均为 REGISTER_KERNEL_BUILDER 宏, 如下图 7.26 注意: 当前只支持注册 MLU 支持的数据类型, 不支持的无法注册; 可以通过 HostMemory 对特定输入进行限制, 如需要在 CPU 上做处理的输入数据。

```

1 // filename: mlu_ops.h
2 struct MLUYolov3DetectionOutputOpParam {
3     // 数据成员声明
4     .....
5     // 构造函数
6     MLUYolov3DetectionOutputOpParam (...) {
7         : batchNum_(batchNum) ,
8         inputNum_(inputNum) ,
9         classNum_(classNum) ,
10        maskGroupNum_(maskGroupNum) ,
11        maxBoxNum_(maxBoxNum) ,
12        netw_(netw) ,
13        neth_(neth) ,
14        confidence_thresh_(confidence_thresh) ,
15        nms_thresh_(nms_thresh) ,
16        inputWs_(inputWs) ,
17        inputHs_(inputHs) ,
18        biases_(biases) {}
19    }
20}
21

```

(a)

```

1 // filename: mlu_ops.h
2 DECLARE_OP_CLASS(mlu...) ;
3 DECLARE_OP_CLASS(MLUYolov3DetectionOutput) ;
4 DECLARE_OP_CLASS(mlu...) ;

```

(b)

图 7.19 Yolov3DetectionOutput 算子 MLUOps 声明

6. 算子注册在 tensorflow/core/ops 目录下找到对应的 Op 注册文件,对于 Yolov3DetectionOutput 为 image_ops.cc 文件。通过这个文件可知 Yolov3DetectionOutput 算子有三个输入和一个输出,三个输入分别用 input0、input1、input2 标识,输出用 predicts 标识,输入输出的数据类型均用 T 表示即该算子的输出与输入数据类型一致。Attr("T: type") 表示 T 允许的数据类型为 type,也就是 TensorFlow 支持的所有数据类型,其余为各超参数配置包括 confidence 阈值以及 NMS 阈值等。

以上 6 个步骤为添加 cnplugin 中 Yolov3DetectionOutput 算子的基本流程,除此之外,为了将该算子最终集成在 TensorFlow 中,编译时还需要在 tensorflow/core/kernels/BUILD 中添加以下信息。

7.1.5.4 DLP 执行

在将 BCL 实现的 Yolov3DetectionOutput 算子成功集成到框架后,需要修改之前转换完成的 int8 pb 模型,在模型中增加 Yolov3DetectionOutput 后处理算子,这一步可以使用编程框架机理章节中介绍的 pb 与 pbtxt 相互转换工具实现。先将 pb 转换成 pbtxt,然后具体在 pbtxt 中添加如下图 7.29所示的节点,最后再转换成 pb: 注意,由于后处理节点已经将三种类型的 feature map 做过一定处理,与 pb 原模型中的操作有一定冗余,为了确保精度的准确性,选择"conv_lbbox/BiasAdd", "conv_mbbox/BiasAdd", "conv_sbbox/BiasAdd" 作为后处理算子的输入节点。至此,只需修改推理代码,将后处理部分注释掉即可完成整个 YOLOv3

```

1 // filename: yolov3detectionoutput.cc
2
3 Status MLUYolov3DetectionOutput::CreateMLUOp(std::vector<MLUTensor*> &inputs, \
4     std::vector<MLUTensor*> &outputs, void *param) {
5     TF_PARAMS_CHECK(inputs.size() > 0, "Missing input");
6     TF_PARAMS_CHECK(outputs.size() > 0, "Missing output");
7     MLUBaseOp *op_ptr = nullptr;
8     MLUTensor* input0 = inputs.at(0);
9     MLUTensor* input1 = inputs.at(1);
10    MLUTensor* input2 = inputs.at(2);
11    MLUTensor* output = outputs.at(0);
12    MLUTensor* buffer = outputs.at(1);
13
14    MLULOG(3) << "CreateYolov3DetectionOutputOp"
15        << ", input0: " << lib::MLUTensorUtil(input0).DebugString()
16        << ", input1: " << lib::MLUTensorUtil(input1).DebugString()
17        << ", input2: " << lib::MLUTensorUtil(input2).DebugString()
18        << ", output: " << lib::MLUTensorUtil(output).DebugString()
19        << ", buffer: " << lib::MLUTensorUtil(buffer).DebugString();
20    int batchNum = ((ops::MLUYolov3DetectionOutputOpParam*)param)->batchNum_;
21    int inputNum = ((ops::MLUYolov3DetectionOutputOpParam*)param)->inputNum_;
22    int classNum = ((ops::MLUYolov3DetectionOutputOpParam*)param)->classNum_;
23    int maskGroupNum = ((ops::MLUYolov3DetectionOutputOpParam*)param)->maskGroupNum_;
24    int maxBoxNum = ((ops::MLUYolov3DetectionOutputOpParam*)param)->maxBoxNum_;
25    int netw = ((ops::MLUYolov3DetectionOutputOpParam*)param)->netw_;
26    int neth = ((ops::MLUYolov3DetectionOutputOpParam*)param)->neth_;
27    float confidence_thresh = ((ops::MLUYolov3DetectionOutputOpParam*)param)->
28        confidence_thresh_;
29    float nms_thresh = ((ops::MLUYolov3DetectionOutputOpParam*)param)->nms_thresh_;
30    int* inputWs = ((ops::MLUYolov3DetectionOutputOpParam*)param)->inputWs_;
31    int* inputHs = ((ops::MLUYolov3DetectionOutputOpParam*)param)->inputHs_;
32    float* biases = ((ops::MLUYolov3DetectionOutputOpParam*)param)->biases_;
33
34    cnmlPluginYolov3DetectionOutputOpParam_t mlu_param;
35    const int num_anchors = 3;
36    cnmlCoreVersion_t core_version = CNML_MLU270;
37
38    // 调用cnmlCreatePluginYolov3DetectionOutputOpParam
39    cnmlCreatePluginYolov3DetectionOutputOpParam(...);
40    std::vector<MLUTensor*> input_tensors = {input0, input1, input2};
41    std::vector<MLUTensor*> output_tensors = {output, buffer};
42
43    TF_STATUS_CHECK(lib::CreateYolov3DetectionOutputOp(...));
44
45    base_ops_.push_back(op_ptr);
46
47    return Status::OK();
}

```

图 7.20 Create 函数实现

模型的在线加速推理工作。

7.1.6 实验评估

相比其它两个综合实验，YOLOv3 实验困难度中等，因此难度系数分为 1.1。学生最终得分为 1.1 乘以相应的得分。

- 60 分标准：补全 nms_detection.h 文件，cnplugin 可正常编过。

```
1 // filename: yolov3detectionoutput.cc
2
3 Status MLUYolov3DetectionOutput::Compute(const std::vector<void *> &inputs,
4     const std::vector<void *> &outputs, cnrtQueue_t queue) {
5     int num_input = inputs.size();
6     int num_output = outputs.size();
7     assert(num_input == 3);
8     assert(num_output == 2);
9     // 调用 ComputeYolov3DetectionOutputOp 进行计算
10    TF_STATUS_CHECK(lib::ComputeYolov3DetectionOutputOp(...));
11
12    // todo, delete sync queue after delay copy
13    cnrtSyncQueue(queue);
14    return Status::OK();
15 }
```

(a)

图 7.21 Compute 函数实现

```
1 Status Yolov3DetectionOutput(OpKernelContext* ctx,
2     Tensor* tensor_input0,
3     Tensor* tensor_input1,
4     Tensor* tensor_input2,
5     int batchNum,
6     int inputNum,
7     int classNum,
8     int maskGroupNum,
9     int maxBoxNum,
10    int netw,
11    int neth,
12    float confidence_thresh,
13    float nms_thresh,
14    int* inputWs,
15    int* inputHs,
16    float* biases,
17    Tensor* output1,
18    Tensor* output2){
19    ops::MLUYolov3DetectionOutputOpParam op_param(
20        batchNum,
21        inputNum,
22        classNum,
23        maskGroupNum,
24        maxBoxNum,
25        netw,
26        neth,
27        confidence_thresh,
28        nms_thresh,
29        inputWs,
30        inputHs,
31        biases);
32    return CommonOpImpl<ops::MLUYolov3DetectionOutput>(
33        ctx,
34        {tensor_input0, tensor_input1, tensor_input2},
35        {output1, output2},
36        static_cast<void*>(&op_param));
37 }
```

图 7.22 Yolov3DetectionOutput 算子 MLUOpKernel 实现

```

1 namespace tensorflow {
2 template<typename T>
3 class MLUYolov3DetectionOutputOp: public MLUOpKernel{
4     public:
5         explicit MLUYolov3DetectionOutputOp(OpKernelConstruction* context):MLUOpKernel(
6             context){
7             OP_REQUIRES_OK(context,context->GetAttr("batchNum",&batchNum_));
8             OP_REQUIRES_OK(context,context->GetAttr("inputNum",&inputNum_));
9             OP_REQUIRES_OK(context,context->GetAttr("classNum",&classNum_));
10            OP_REQUIRES_OK(context,context->GetAttr("maskGroupNum",&maskGroupNum_));
11            OP_REQUIRES_OK(context,context->GetAttr("maxBoxNum",&maxBoxNum_));
12            OP_REQUIRES_OK(context,context->GetAttr("netw",&netw_));
13            OP_REQUIRES_OK(context,context->GetAttr("neth",&neth_));
14            OP_REQUIRES_OK(context,context->GetAttr("confidence_thresh",&
15                confidence_thresh_));
16            OP_REQUIRES_OK(context,context->GetAttr("nms_thresh",&nms_thresh_));
17            OP_REQUIRES_OK(context,context->GetAttr("inputWs",&inputWs_));
18            OP_REQUIRES_OK(context,context->GetAttr("inputHs",&inputHs_));
19            OP_REQUIRES_OK(context,context->GetAttr("biases",&biases_));
20        }
21
22        void ComputeOnMLU(OpKernelContext* context) override {
23            auto* stream = context->op_device_context()->mlu_stream();
24            auto* mlustream_exec =
25                context->op_device_context()->mlu_stream()->parent();
26            //参数检查与处理
27            .....
28        }
29
30    };
31
32    void ComputeOnMLU(OpKernelContext* context) override {
33        auto* stream = context->op_device_context()->mlu_stream();
34        auto* mlustream_exec =
35            context->op_device_context()->mlu_stream()->parent();
36        //参数检查与处理
37        .....
38    }
39
40    void ComputeOnCPU(OpKernelContext* context) override {
41        auto* stream = context->op_device_context()->cpu_stream();
42        auto* mlustream_exec =
43            context->op_device_context()->cpu_stream()->parent();
44        //参数检查与处理
45        .....
46    }
47
48    void ComputeOnGPU(OpKernelContext* context) override {
49        auto* stream = context->op_device_context()->gpu_stream();
50        auto* mlustream_exec =
51            context->op_device_context()->gpu_stream()->parent();
52        //参数检查与处理
53        .....
54    }
55
56    void Compute(OpKernelContext* context) override {
57        if (output->NumElements() > 0 && buffer->NumElements() > 0 ) {
58            OP_REQUIRES_OK(
59                context,
60                stream->Yolov3DetectionOutput(
61                    context, input0, input1, input2, batchNum_, inputNum_,
62                    classNum_, maskGroupNum_, maxBoxNum_, netw_, neth_,
63                    confidence_thresh_, nms_thresh_, inputWs_.data(), inputHs_.data(),
64                    biases_.data(), output, buffer));
65        } else {
66            mlustream_exec->insert_unsupported_op(context, op_parameter);
67        }
68    }
69
70    private:
71        int batchNum_;
72        .....
73    };
74}

```

图 7.23 Yolov3DetectionOutput 算子 MLUOpKernel 实现

```

1 //输出形状推断及输出内存分配
2 .....

```

图 7.24 Yolov3DetectionOutput 算子 MLUOpKernel 实现

```

1 //调用MLUStream层接口完成算子计算
2 if(output->NumElements() > 0 && buffer->NumElements() > 0 ){
3     OP_REQUIRES_OK(
4         context,
5         stream->Yolov3DetectionOutput(
6             context, input0, input1, input2, batchNum_, inputNum_,
7             classNum_, maskGroupNum_, maxBoxNum_, netw_, neth_,
8             confidence_thresh_, nms_thresh_, inputWs_.data(), inputHs_.data(),
9             biases_.data(), output, buffer));
10    } else {
11        mlustream_exec->insert_unsupported_op(context, op_parameter);
12    }
13
14    private:
15        int batchNum_;
16        .....
17    };
18}

```

图 7.25 Yolov3DetectionOutput 算子 MLUOpKernel 实现

- 70 分标准：在 60 分的基础上完成 TensorFlow 的集成编译，完成 pb 模型添加后处理大算子的操作，执行测试脚本时 map 值高于 30%，单 batch 延时（包含后处理）低于 300ms。

```

1 #if CAMBRICON_MLU
2 #include "tensorflow/core/kernels/yolov3_detection_output_op_mlu.h"
3 namespace tensorflow {
4 #define REGISTER_MLU(T) \
5   REGISTER_KERNEL_BUILDER( \
6     Name("Yolov3DetectionOutput") \
7     .Device(DEVICE_MLU) \
8     .TypeConstraint<T>("T"), \
9     MLUYolov3DetectionOutputOp<T>); \
10 TF_CALL_MLU_FLOAT_TYPES(REGISTER_MLU);
11 #undef REGISTER_MLU
12 #endif // CAMBRICON_MLU
13 }

```

图 7.26 Yolov3DetectionOutput 算子注册

```

1 REGISTER_OP("Yolov3DetectionOutput")
2   .Output("predicts: T")
3   .Input("input0: T")
4   .Input("input1: T")
5   .Input("input2: T")
6   .Attr("batchNum: int")
7   .Attr("inputNum: int")
8   .Attr("classNum: int")
9   .Attr("maskGroupNum: int")
10  .Attr("maxBoxNum: int")
11  .Attr("netw: int")
12  .Attr("neth: int")
13  .Attr("confidence_thresh: float")
14  .Attr("nms_thresh: float")
15  .Attr("inputWs: list(int) = [13, 26, 52]")
16  .Attr("inputHs: list(int) = [13, 26, 52]")
17  .Attr("biases: list(float) = [116, 90, 156, 198, 373, 326, 30, 61, 62, 45, 59, 119,
18    10, 13, 16, 30, 33, 23]")
19  .Attr("T: type")
20  .SetShapeFn([](InferenceContext *c){
21    return SetOutputForYolov3DetectionOutput(c);
22  });

```

图 7.27 Yolov3DetectionOutput 算子注册

- 80 分标准：在 70 分的基础上，执行测试脚本时 map 值高于 50%，单 batch 延时（包含后处理）低于 100ms。
- 90 分标准：在 80 分的基础上，执行测试脚本时 map 值高于 54%，单 batch 延时（包含后处理）低于 50ms。
- 100 分标准：在 90 分的基础上，执行测试脚本时 map 值高于 56%，单 batch 延时（包含后处理）低于 25ms。

7.1.7 实验思考

1.NMS BCL 相比 CPU 实现有何优势？2. 有哪些方法可以提升 BCL 算子的性能？

```

1 .....
2 cc_library(
3     name = "image",
4     deps = [
5         ":adjust_contrast_op",
6         .....
7         ":yolov3_detection_output_op",
8     ],
9 )
10 .....
11 tf_kernel_library(
12     name = "yolov3_detection_output_op",
13     prefix = "yolov3_detection_output_op",
14     deps = [
15         "//tensorflow/core:core_cpu",
16         "//tensorflow/core:framework",
17         "//tensorflow/core:lib",
18         "//tensorflow/core:lib_internal",
19         "//third_party/eigen3",
20         ] + if_mlu([
21             "//tensorflow/stream_executor:mlu_stream_executor"]),
22 )

```

图 7.28 TensorFlow 编译

```

1 node {
2     name: "Yolov3DetectionOutput"
3     op: "Yolov3DetectionOutput"
4     input: "conv_bbbox/BiasAdd"
5     input: "conv_mbbox/BiasAdd"
6     input: "conv_sbbox/BiasAdd"
7     attr {
8         key: "T"
9         value {
10             type: DT_FLOAT
11         }
12     }
13     ...
14     attr {
15         key: "nms_thresh"
16         value {
17             f: 0.449999988079
18         }
19     }
20 }
21 versions {
22     producer: 38
23 }

```

图 7.29 YOLOv3 增加后处理节点

```
1 #yolov3-release/evaluate.py
2 def predict_bak(self, images):
3
4     org_h = [0 for i in range(self.batch_size)]
5     org_w = [0 for i in range(self.batch_size)]
6     for i in range(self.batch_size):
7         org_h[i], org_w[i], _ = images[i].shape
8
9     image_data = utils.images_prepocess(images, [self.input_size, self.input_size])
10
11    start = time.time()
12    pred_sbbox, pred_mbbox, pred_lbbox = self.sess.run(
13        [self.pred_sbbox, self.pred_mbbox, self.pred_lbbox],
14        feed_dict={
15            self.input_data: image_data,
16            #self.trainable: False
17        }
18    )
19    np.savetxt("pred_sbbox.txt", pred_sbbox.flatten())
20    np.savetxt("pred_mbbox.txt", pred_mbbox.flatten())
21    np.savetxt("pred_lbbox.txt", pred_lbbox.flatten())
22    end = time.time()
23
24    batch_bboxes = []
25    for idx in range(self.batch_size):
26        pred_bbox = np.concatenate([np.reshape(pred_sbbox[idx], (-1, 5 + self.
27        num_classes)),
28                                    np.reshape(pred_mbbox[idx], (-1, 5 + self.
29        num_classes)),
30                                    np.reshape(pred_lbbox[idx], (-1, 5 + self.
31        num_classes))], axis=0)
32        bboxes = utils.postprocess_boxes(pred_bbox, (org_h[idx], org_w[idx]), self.
33        input_size, self.score_threshold)
34        batch_bboxes.append(utils.nms(bboxes, self.iou_threshold))
35    print("bbox num : ", len(batch_bboxes))
36    for i in range(len(batch_bboxes)):
37        print(batch_bboxes[i])
38    exit(0)
39    return batch_bboxes, (end - start)
```

图 7.30 原始推理

```

1 # yolov3-bcl/demo/evaluate.py
2 def predict(self, images):
3     .....
4     start = time.time()
5     bbox_raw = self.sess.run(
6         self.bbox_raw,
7         feed_dict={
8             self.input_data: image_data,
9             #self.trainable: False
10        }
11    )
12    end = time.time()
13    batch_bboxes = []
14    num_batches = 1
15    num_boxes = 1024 * 2
16    predicts_mlu = bbox_raw.flatten()
17    print("org_h[0], ",org_h[0])
18    print("org_w[0], ",org_w[0])
19    for batchIdx in range(num_batches):
20        result_boxes = int(predicts_mlu[batchIdx * (64 + num_boxes * 7)])
21        current_bboxes = []
22        print("result_boxes : ",result_boxes)
23        print("x1, y1, x2, y2, score, classId")
24        for i in range(result_boxes):
25            # batchId, classId, score, x1, y1, x2, y2
26            batchId = predicts_mlu[i * 7 + 0 + 64 + batchIdx * (64 + num_boxes * 7)]
27            classId = predicts_mlu[i * 7 + 1 + 64 + batchIdx * (64 + num_boxes * 7)]
28            score = predicts_mlu[i * 7 + 2 + 64 + batchIdx * (64 + num_boxes * 7)]
29            x1 = predicts_mlu[i * 7 + 3 + 64 + batchIdx * (64 + num_boxes * 7)
30            ] * org_w[0]
31            y1 = predicts_mlu[i * 7 + 4 + 64 + batchIdx * (64 + num_boxes * 7)
32            ] * org_h[0]
33            x2 = predicts_mlu[i * 7 + 5 + 64 + batchIdx * (64 + num_boxes * 7)
34            ] * org_w[0]
35            y2 = predicts_mlu[i * 7 + 6 + 64 + batchIdx * (64 + num_boxes * 7)
36            ] * org_h[0]
37            print(x1, y1, x2, y2, score, classId)
38            # bbox = [xmin, ymin, xmax, ymax, score, class]
39            bbox = [x1, y1, x2, y2, score, classId]
40            current_bboxes.append(np.array(bbox))
41    batch_bboxes.append(current_bboxes)
42    print("bbox num : ",len(batch_bboxes))
43    for i in range(len(batch_bboxes)):

```

图 7.31 添加后处理大算子

7.2 文本检测 OCR-EAST

7.2.1 实验目的

本实验主要完成文本检测的代表性算法——EAST 网络移植到智能处理器 DLP 上，并进行性能优化，使读者可以借助 DLP 处理器完成完整的文本检测任务。

因此，本实验的主要目的包括：

1. 通过使用 Tensorflow 在 CPU 和智能处理器 DLP 上进行在线推理，掌握使用 Tensorflow 编写文本检测应用；
2. 通过用智能编程语言实现 EAST 网络中的 Split+sub+concat 合并算子，深入掌握智能编程语言的算子开发和优化方法；
3. 通过在 Tensorflow 中添加合并算子，代替单算子，掌握在 Tensorflow 框架中添加合并算子的方法并优化性能；

使用合并算子代替多个单算子是 DLP 的一种常见性能优化手段。由于多个单算子执行时，每个算子都需要完成一次片上内存调度和相应计算，所以往往执行效率有较大提升空间。当多个算子合并成大算子之后芯片可以在一次数据调度之后就完成多步计算，减少整体数据搬运的成本，从而提升性能。

实验时间预计为两周。

7.2.2 背景介绍

7.2.2.1 OCR



图 7.32 OCR 应用举例

从传说中的仓颉造字以来，文字或文本就为人类的发展进步带来了重要贡献。到现在为止，文本依旧是包含丰富而准确信息的一个大类场景，因此自然场景中的文本检测已成为计算机视觉和文档分析中重要且活跃的研究主题。尤其是近年来，尽管仍然存在各种挑战（例如，噪声，模糊，失真，遮挡和变化），但社区在这些领域的研究工作激增并取得了实质性进展。

在实际应用中，在新闻出版，邮政快递，金融服务等领域都广泛使用了文本识别算法。例如对于如图 7.32 所示的针对会计领域的发票文本识别就大大减轻了人工比对的负担。

光学字符识别（Optical Character Recognition,OCR）是指对文本资料的图像文本进行文

字识别, 获取文本资料版面信息的过程。OCR 算法的研究有很长的历史, 在近一些年基于各种深度学习的算法逐渐成为重要的研究方向。

OCR 过程大致可以分为: 图像获取-> 图像预处理-> 文字检测-> 文字识别-> 输出。

7.2.2.2 EAST

EAST 算法全名是 Efficient and Accuracy Scene Text, 它提出了一个快速而准确的轻量级场景文本检测 pipeline, 解决大多数文本检测算法准确性和效率缺陷。该 Pipeline 只有两个阶段: 第一阶段是基于 FCN 模型的文本框检测; 第二阶段是对生成的文本框 (旋转或矩形) 经过非极大值抑制得到最终结果, 从而免去其他中间步骤, 实现端到端训练。

- 网络结构

EAST 的网络结构分为三大部分, 如图 7.33 所示:

第一部分:Feature extractor stem(PVANet)

该部分是主干特征检测网络, 引入多尺度检测的思想, 提取不同尺寸卷积核下的特征并用于后期的特征组合, 以适应文本行尺度的变化。作者采用了 PVANet 模型作为主干网络, 实际使用的时候也可以采用 VGG16 或者 Resnet;

第二部分:Feature-merging branch

该部分运用了 Unet 的思想, 主要是合并第一部分提取的特征, 通过池化和 Concat 来恢复尺寸。

其中, 特征合并主要通过逐层合并的方式, 首先将第一部分对应的特征图输入到一个池化层恢复尺度, 然后与当前层特征图进行 Concat, 再通过 1×1 卷积来减少通道数, 最后利用 3×3 卷积将局部信息融合以最终产生该合并阶段的输出;

第三部分:Output Layer

该部分输出分为 3 类:score_map, RBOX geometry, QUAD geometry:

Score_map 代表此处是否有文字的可能性;

RBOX geometry 中的五个通道, 分别代表每个像素点到文本框边线的距离, 加上一个文本框的旋转角;

QUAD geometry 中的八个通道, 分别代表着每个像素点到文本框任意四边形的 xy 距离。

- 后处理——局部感知 NMS

EAST 的后处理与通用目标检测类似, 需要将网络的结果经过非极大值抑制 (NMS) 来取得最终结果。由于 EAST 的网络输出是成千上万个几何文本框, 需要根据阈值过滤, 然后将文本框的置信度得分加权平均, 得到合并后的文本框坐标。

将经过合并后的文本框经过通用的 NMS 处理, 详情可参考 7.1 章节, 得到最终结果。

7.2.2.3 Split 算子, Sub 算子和 Concat 算子

在 EAST 网络中需要调用 Split 算子, Sub 算子和 Concat 算子完成两个张量之间的相减操作。

Split 算子可以将某个张量在某个维度上进行切分。从而将某个规模较大的张量切分为

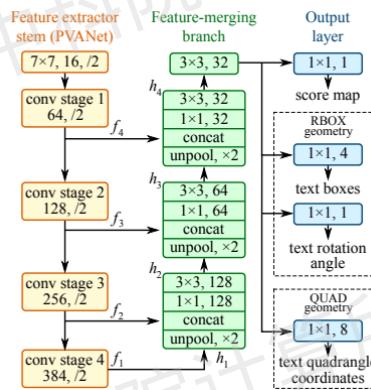


图 7.33 east 网络结构

多个规模较小的张量。Sub 算子可以完成两个张量之间的对位减操作。Concat 算子可以将多个较小规模张量合并成一个较大规模张量。当这三个算子组合运行时实现了将某个较大张量切分后，各自与较小张量相减，最后再将相减结果合并的操作。

在此过程中可以发现 Split 和 Concat 操作均为单纯的内存搬运操作，实际的计算内容只有 Sub 的部分。这提示我们是否可以通过一些优化手段减少这种数据搬运。而这种优化技巧也是本次实验的核心内容。

7.2.3 实验环境

本实验所涉及的硬件平台和软件环境如下：

- 硬件平台：硬件平台基于前述的 DLP 云平台环境。
- 软件环境：所涉及的 DLP 软件开发模块包括编程框架 TensorFlow、高性能库 CNML、运行时库 CNRT、编程语言及编译器、运行时库 CNRT。

7.2.4 实验内容

本实验主要是在 DLP 硬件上实现并优化以 EAST 网络模型为核心的目标检测应用。针对用户从网络上下载的标准 EAST 模型，经过模型量化并配置 DLP 相关参数，则可以采用 DLP 上的定制 TensorFlow 版本来运行。为了充分发挥 DLP 的计算能力，进一步采用智能编程语言 BCL 实现 EAST 网络模型中的 Split+Sub+Concat 合并算子，并将其添加至 TensorFlow 框架中，代替原有的 Split、Sub、Concat 单算子。具体实验内容包括：

1. EAST CPU 运行：利用标准的与训练 CKPT 模型，通过定制的 TensorFlow 版本在 CPU 上运行 EAST，得到 Float16 精度的 PB 模型；
2. EAST DLP 运行：将步骤 1 得到的 Float16 精度的 PB 模型进行 INT8 量化后，通过定制的 TensorFlow 版本在 DLP 硬件上运行；
3. Split+Sub+Concat 合并算子的 BCL 实现：采用 BCL 实现的 Split+Sub+Concat 合并运算；
4. 框架算子集成：将用 BCL 实现的 Split+Sub+Concat 合并算子集成到 TensorFlow 框架中。

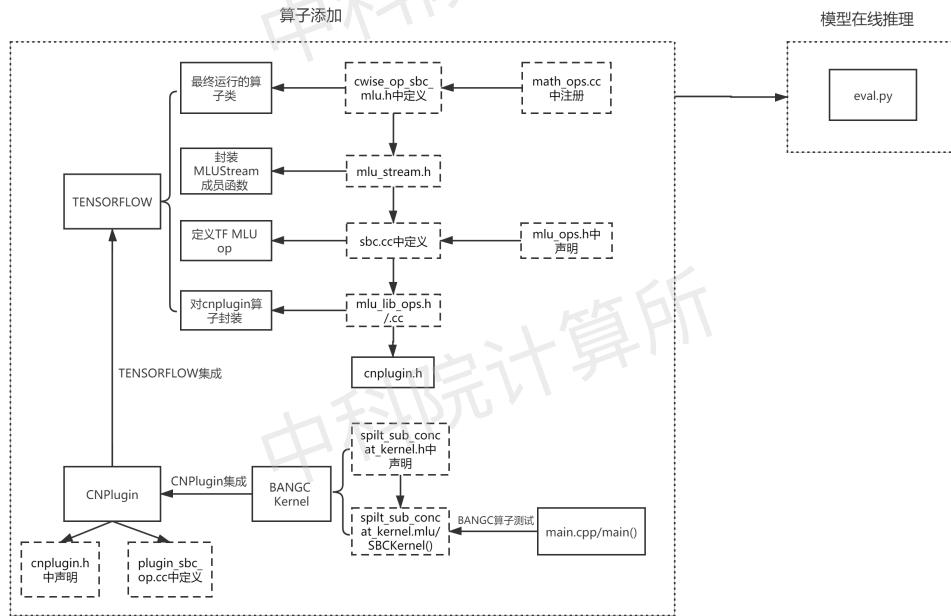


图 7.34 实验内容

如图7.34，图中虚线框内容为本次实验需要修改的文件。主要实验流程为 BCL 实现 Split+Sub+Concat 并集成入 Tensorflow 推理框架中。

7.2.5 实验步骤

如前所述,详细的实验步骤主要包括:EAST CPU 运行、EAST DLP 运行、Split+Sub+Concat 合并算子的 BCL 实现、框架算子集成等。

7.2.5.1 EAST CPU 运行

为了使得标准的 EAST 网络模型在 DLP 上运行,需要通过运行 CPU 实现,将 CKPT 预训练模型转换成 Float16 精度的 PB 模型。具体步骤如下:

1. 获取开源数据集和 CKPT 模型:<https://github.com/argman/EAST>
2. 按照以下目录结构配置数据集和模型:

数据集目录结构:

```

1 /home/Cambricon-MLU270/datasets/tensorflow_models/east
2     |-- 2015Challenge4_Test_Task1_GT/
3         |-- 2015ch4_test_images/
4             |-- 2015ch4_training_images/
5             |-- 2015ch4_training_localization_transcription_gt/

```

图 7.35 EAST 数据集目录结构

模型目录结构:

CPU 运行之前,需要 MODEL_PATH 指定到 east_icdar2015_resnet_v1_50_rbox 下, DATASET_PATH

```

1 /home/Cambricon-MLU270/models/tensorflow_models/east/east_icdar2015_resnet_v1_50_rbox
2     |-- checkpoint
3     |-- model.ckpt-49491.data-00000-of-00001
4     |-- model.ckpt-49491.index
5     |-- model.ckpt-49491.meta

```

图 7.36 EAST 模型目录结构

指定到 2015ch4_test_images 下,GT_PATH 指定到 2015Challenge4_Test_Task1_GT 下。

3. DLP 软件环境中已经适配了 EAST CPU 运行环境和 DLP 运行环境, 需要进入 tensorflow_models/cambricon_examples/EAST/ 目录下;

4. 安装推理所需软件包: 一些必备的 Python 安装包在所提供的云平台软件环境中已经安装完成, 但对于每个综合实验的特有依赖还需进行单独安装, 使用 pip 命令完成依赖的安装与升级: pip install -r requirements.txt

5. 模型推理和转换: 执行如下命令完成在 CPU 上的推理, 并将 CKPT 模型转换为 Float16 精度的 PB 模型: ./run_cpu.sh

6. 整理模型路径: 生成的 PB 模型在./cpu_pb 中, 需要将得到的 east.pb 拷贝至/home/Cambricon-MLU270/models/tensorflow_models/EAST/east.pb

7.2.5.2 EAST DLP 运行

下面介绍如何将 7.2.5.1 小节得到的 Float16 精度 PB 模型量化为 INT8 精度的 PB 模型。DLP 软件环境提供 fppb_to_intpb 工具来进行模型量化。

1. 进入 tensorflow/cambricon_examples/tools/fppb_to_intpb 目录;
2. 生成数据文件: 按照模型所需要的数据集以及数据集的具体路径来修改 generate_image_list.py 文件中的 image_libs_map, 具体如图 7.37 所示, 来生成相应的数据文件。

```

1 // generate_image_list.py
2
3 import os
4 from os.path import isfile, join
5
6 max_image_count = 100
7 image_libs_map = {
8     "east": join(os.environ.get("TENSORFLOW_MODELS_DATA_HOME"), "east/2015"
9                  "ch4_training_images")
10 }
11 if __name__ == "__main__":
12     for image_lib, image_path in image_libs_map.items():
13         n = 0
14         with open("image_list_{}".format(image_lib), "w") as image_list:
15             for root, dirs, files in os.walk(image_path):
16                 for f in files:
17                     if n >= max_image_count:
18                         break
19                     if f.endswith("jpg"):
20                         image_list.write(join(image_path, f) + "\n")
21                         n = n + 1

```

图 7.37 EAST 量化生成数据文件

3. 生成量化配置文件: 在进行量化前需设置相应的配置文件以指定量化相关的参数。DLP 软件环境提供了 generate_ini.py 来生成指定模型的量化配置文件。针对 EAST, 需要将 generate_ini.py 中的 model_name 进行修改, 即指定要生成的配置文件是针对 EAST 的:models_name = ['EAST']

执行 DLP 定制化的 PB 模型量化脚本 generate_ini.py 后会生成 config 文件夹, 其中包含了 EAST_naive_int8.ini 参数配置文件, 该配置文件包含如图7.38所示的信息。同时为了规范模型路径, 需要修改 save_model_path;

```

1 // EAST_naive_int8.ini
2
3 [preprocess]
4 mean = 0, 0, 0      #均值,顺序依次为_mean_r、_mean_g、_mean_b
5 std = 1.0           #方差
6 color_mode = rgb    #网络的输入图片是rgb还是bgr
7 crop = 544, 544     #前处理最终将图片处理为416×416大小
8 calibration = east_preprocess_cali   #校准数据读取及前处理的方式,可以根据需求进行自定义,[preprocess]和[data]中定义的参数均为calibration的输入参数
9
10 [config]
11 activation_quantization_alg = naive    #输入量化模式,可选naive和threshold_search,naive为基础模式,threshold_search为阈值搜索模式
12
13 device_mode = clean                   #可选clean、mlu和origin,建议使用clean,使用clean
                                         #生成的模型在运行时会自动选择可运行的设备
14
15 use_convfirst = False                 #是否使用convfirst
16 quantization_type = int8            #量化位宽,目前可选int8和int16
17 debug = False                       #是否为debug模式
18 weight_quantization_alg = naive    #权值量化模式,可选naive和threshold_search,naive为基础模式,threshold_search为阈值搜索模式
19
20 int_op_list = Conv, FC, LRN        #要量化的layer的类型,目前只能量化Conv、FC和
                                         #LRN
21 channel_quantization = False       #是否使用分通道量化,目前不支持为True
22
23 [model]
24 output_tensor_names = feature_fusion/Conv_7/Sigmoid:0, feature_fusion(concat_3:0) #输出
                                         #Tensor的名字,可以是多个,以逗号隔开
25 original_models_path = /home/Cambricon-MLU270/models/tensorflow_models/east/east.pb #输入
                                         #pb
26 save_model_path = /home/Cambricon-MLU270/models/tensorflow_models/east/east_int8.pb    #
                                         #输出pb
27 input_tensor_names = input_images:0          #输入Tensor的名字,可以是多个,以
                                         #逗号隔开
28
29 [data]
30 num_runs = 2                         #运行次数

```

图 7.38 EAST 量化配置文件

而提高数据集整体的量化质量。但是对于不存在异常值, 数据分布紧凑的情况下, 不建议使用该算法, 比如网络权值的量化。其中 device_mode 可以设置输出 pb 所有节点的 device, 有三种配置方式:clean、mlu 和 origin。其中, mlu 将输出 pb 的所有节点的 device 都设置为 MLU, 即都在 MLU 上运行; clean 将输出 pb 的所有节点的 device 清除, 运行时根据算子注册情况自动选择可运行的设备; origin 则使用和输入 pb 一样的设备指定。

4. 执行命令: `python fppb_to_intpb.py` 完成模型量化最终得到 `east_int8.pb`.

完成 CPU 侧的推理后, 接下来需要将其移植在 DLP 上, 以加快其推理速度。此时我们只需仿照第 7 章编程语言算子实验中的 DLP 推理移植部分。DLP 软件栈中已经完成了 EAST 的移植, 只需要将 `MODEL_PATH` 指定到 `east_int8.pb`, 执行得到精度和性能数据:

```
./run.sh 16 MLU270 int8 100 1
```

7.2.5.3 BANGC 代码实现

1. 设计原则

SBC 合并算子实现在 `spilt_sub_concat_kernel.h` 和 `spilt_sub_concat_kernel.mlu`, 分别对算子进行定义和实现。在使用 BCL 设计 Split + Sub + Concat 合并算子的时候, 需要考虑以下 BCL 设计原则:

(a) 能够处理的数据来源: GDRAM, NRAM, SRAM 中的一种

(b) 能够将数据存放到: GDRAM, NRAM, SRAM 中的一种

(c) 每个 Core 都有一块片上存储空间 NRAM, 其大小约为 512KB

(d) 每个 Cluster 上分配一块 SRAM, 其大小 2MB, 并且单核任务不支持 SRAM 的使用, UNION1 任务和单核任务不支持 SRAM 通信 UNION2 以上任务才支持 SRAM 通信

保证算子正确性的情况下, 需要考虑以下四种优化方向:

(a) 对访存进行优化以充分利用片上存储, 比如 NRAM 复用

(b) 对计算逻辑进行优化以充分削减计算量, 充分利用张量运算器

(c) 充分利用多核并行 (计算任务拆分) 来提升并行度以及隐藏访存延迟

(d) 充分利用编译器提供的各种编译优化选项, 比如 O2 或者 O3 编译优化

2. 设计原理

首先, 我们需要分析输入数据的规模, 通过开源模型可视化软件 Netron 查看 PB 模型, 可以确定整个网络模型的结构, 并在 DLP MLU 执行过程中, 打印出 `input_images` 的输出维度为 (1,672,1280,3), 即是 Split 的输入维度;

由于 BCL 使用的数据类型是 Half 类型, 占据 2 个字节, 所以计算出输入规模是 $1 \times 672 \times 1280 \times 3 \times 2 / 1024 = 5040$, 在充分利用多核并行的情况下, 将数据分块到每个 core, 每块 NRAM 为 $5040 / 16 = 315KB$, 在 512KB 的限制内;

使用 BangC 实现 Split 的时候, 需要注意, 输入数据的摆数是 NHWC, C 方向的数据是连续的。如果在 C 方向进行 Split, 则需要先转置为 NCHW, 或者使用 `_bang_maskmove` 来进行 Split, 再去做 Sub, 这样会增加计算耗时或者 NRAM 的使用。而最简洁的方法是使用 `_bang_cycle_sub`, 对 C 方向进行 cyclesub, 省略掉了 Split 和 Concat 部分, 大大提升效率, 减少 NRAM 的使用。整体设计流程图如图 7.39 所示:

下面以图 7.40 中的代码为例子进行说明, 介绍 Split+Sub+Concat 合并算子的接口介绍:

`input_data_`: 输入数据存放的数据地址;

`output_data_`: 输出数存放的数据地址;

`batch_num_`: Batchsize 的规模。

3. 实现解析

代码实现分为两部分: Kernel 函数实现的 `spilt_sub_concat_kernel.mlu` 和 cnrt 实现的 `main.cpp`



图 7.39 SBC 运行流程

```

1 // spilt_sub_concat_kernel.h
2 __mlu_entry__ void SBCKernel(half* input_data_ ,
3 half* output_data_ ,
4 int batch_num_)
5
  
```

图 7.40 Split+Sub+Concat 合并算子接口

7.2.5.4 Tensorflow 算子添加

本节介绍如何将实现的 BANGC kernel 函数集成至 Tensorflow, 为 EAST 添加 Split+Sub+Concat 合并算子. 添加 MLU 算子主要分为以下几个步骤, 步骤之间有承接关系:

1. CNPlugin 编译

如前所述,CNML 通过 PluginOp 相关接口提供了用户自定义算子和高性能库已有算子协同工作（如算子融合）的机制. 因此, 在完成 Split+Sub+Concat 算子的开发后, 可以利用 PluginOp 相关接口封装出方便用户使用的接口（主要包括 PluginOp 的创建、计算和销毁等接口）, 使得用户自定义算子和高性能库已有算子有一致的编程接口和模式.

图7.43给出了 PluginOp 接口封装的部分示例代码, 主要包括 Create、Compute 和 Destroy 三类函数的具体实现。代码位于 plugin_sbc_op.cc。

-creat 函数:

-Compute 函数:

-Destroy 函数:

2. 算子注册

算子注册在 tensorflow/core/ops 目录下找到对应的 Op 注册文件为 math_ops.cc,: 对于

```

1 // spilt_sub_concat_kernel.mlu
2 #define HWC_SPLIT (((HEIGHT*WIDTH/16) - 1) / ALIGN_SIZE + 1) * ALIGN_SIZE)*CHANNELS
3 #define CHANNELS 3
4 #define HEIGHT 672
5 #define WIDTH 1280
6 #define ALIGN_SIZE 64
7 #define DATA_COUNT ((CHANNELS) * (WIDTH) * (HEIGHT))
8
9 #include "mlu.h"
10
11 __mlu_entry__ void SBCKernel(half* input_data_, half* output_data_, int batch_num_){
12
13     int batch_num = batch_num_;
14
15     // struct timeval start;
16     // struct timeval end;
17     // gettimeofday(&start, NULL);
18
19     __nram__ half split_sub_concat[HWC_SPLIT];
20     __nram__ half tmp0[192];
21
22     // 多核拆分
23     .....
24
25     // 循环创建 cycle_sub mask
26     for (int i =0;i<192;i++){
27         .....
28     }
29
30     for (int i = 0; i<batch_num; i++){
31         for (int j = 0; j < core_loop; j++){
32             // 数据拆分至每个 Core 的 NRAM
33             .....
34             // cycle_sub 代替 split+sub
35             .....
36             // 按顺序拷贝回GDRAM
37             .....
38         }
39         __sync_all();
40     }
41
42     // 计算耗时
43     // gettimeofday(&end, NULL);
44     // uint32_t time_usec = (uint32_t)end.tv_usec - (uint32_t)start.tv_usec;
45     // printf("Hardware Total Time: %u us\n", time_usec);
46     // printf("batch_size: %d us\n", batch_num);
47     // printf("core_num: %d us\n", taskDim);
48
49 }

```

图 7.41 EAST BANGC 算子实现

Split+Sub+Concat 合并算子, 在 `math_ops.cc` 中注册. 如图7.45, Split+Sub+Concat 合并算子包含一个输入节点 `input` 和一个输出节点 `output`, 输入输出的数据类型均用 `T` 表示即该算子的输出与输入数据类型一致. 其中, `Attr("T:type")` 表示 `T` 允许的数据类型为 `type`, 也就是 TensorFlow 支持的所有数据类型.

3. MLUlib

MLUlib 层主要是对 CNML 和 CNRT 接口的封装, 该部分代码在 `tensorflow/stream_executor/mlu/mlu_`

目录下,需要注意的是 MLULIB 层禁止引用 TensorFlow 的头文件,目前只使用了 tensorflow::Status 类方便做返回值处理。添加 Split+Sub+Concat 合并算子需要对该目录下的 mlu_ops.cc 和 mlu_ops.h 文件进行修改,分别对应实现和定义。示例代码如图7.46所示。

4. MIUops

MLUOps 层主要是使用 MLULib 层封装好的 API 完成算子的实现,代码文件在 tensorflow/stream_executor/mlu/mlu_api/ops 下,需要在目录下的 mlu_ops.h 文件中添加新算子类的声明,如图7.47所示。

同时,需要新建一个 sbc.cc 文件来添加 Split+Sub+Concat 合并算子的实现,其中包含 Create 方法和 Compute 方法,如图7.48所示。

Create 方法的职责是将创建好的 baseop 指针调用 base_ops_.push_back(op_ptr); 储存起来;Compute 方法的职责在于调用 lib 层的 compute 函数进行计算,目前算子实现为同步方式,因此需要调用 SyncQueue。

5. MLUStream

MLUStream 层负责 MLULoops 算子类的实例化,其接口都定义在 tensorflow/stream_executor/mlu/mlu 中。可以把 MLUStream 层的算子分为 2 类:通用模版算子与特例化算子。对于 Split+Sub+Concat 合并算子来说,其属于通用模版算子。

通用模版算子满足以下三个条件:MLU 算子的所有输入均来自 OpKernelContext;MLU 算子的所有输出顺序须与 OpkernelContext 的输出顺序一致;MLUTensor 可以被 CreateMLUTensorFromTensor 创建,即 MLUTensor 的形状、数据类型与 TensorFlowtensor 一致。如图7.49

6. MLUOpKernel

MLUOpKernel 层负责对算子抽象定义,其继承了 OpKernel 类,其使用方法与 OpKernel 基本一致。对于每个 MLU 算子均需要实现其构造函数与 ComputeOnMLU 方法。MLUOpKernel 实现的主要功能有参数检查、参数处理、输出形状推断及输出内存分配、调用 MLUStream 层接口完成算子计算。需要在 cwise_op_sbc_mlu.cc 文件中进行注册和 cwise_op_sbc_mlu.h 实现。在下图7.50和7.51中的代码分别显示了这四个不同的步骤

7. BUILD 为了将该算子最终集成在 TensorFlow 中,编译时还需要在 tensorflow/core/kernels/BUILD 中添加以下信息。

7.2.5.5 DLP 执行

在将 BCL 实现的 Split+Sub+Concat 合并算子成功集成到框架后,需要修改之前转换完成的 int8pb 模型,在模型中增加 Split+Sub+Concat 合并算子,这一步可以使用编程框架机理章节中介绍的 pb 与 pbtxt 相互转换工具实现。具体在 pbtxt 中用 Split+Sub+Concat 节点7.52替换原生 Split、Sub、Concat 节点7.53

7.2.6 实验评估

相比其它两个综合实验, EAST 实验困难度较低,因此难度系数分为 1.0。学生最终得分为 1.0 乘以相应的得分。

- 60 分标准: 完成 BANGC 算子的实现与 CNRT 测试, CNRT 测试时延时低于 30ms。

- 70 分标准：在 60 分的基础上完成 TensorFlow 的算子集成，包括 cnplugin 集成与 TensorFlow 的编译，完成 pb 模型的修改操作。执行测试脚本时，修改后的模型精度与修改前误差在 5% 以内，且延时不高于原始的模型。
- 80 分标准：在 70 分的基础上，执行测试脚本时，修改后的模型精度与修改前误差在 1% 以内，且延时至少优于原始的模型 10ms。
- 90 分标准：在 80 分的基础上，执行测试脚本时，修改后的模型精度与修改前误差在 0.1% 以内，且延时至少优于原始的模型 25ms。
- 100 分标准：在 90 分的基础上，执行测试脚本时，修改后的模型精度于修改前完全一致，且延时至少优于原始的模型 40ms。

7.2.7 实验思考

1. 为什么 BCL 算子相比 Split+Sub+Concat CNML 单算子有显著优势？

```

1 // main.cpp
2 ...
3 #include "macro.h"
4 #include "cnrt.h"
5 #include "utils.h"
6
7 #define CHANNELS 3
8 #define HEIGHT 672
9 #define WIDTH 1280
10 #define BATCH_SIZE 1
11 #define DATA_COUNT ((CHANNELS) * (WIDTH) * (HEIGHT))
12
13 using namespace std;
14 typedef unsigned short half;
15
16 extern "C" {
17     void SBCKernel(half* input_data_, half* output_data_, int batch_num_, int core_num_);
18 }
19
20 int main() {
21     const int data_count = DATA_COUNT*BATCH_SIZE;
22     int batch_num_ = BATCH_SIZE;
23     int core_num_ = NUM_MULTICORE;
24     const int channels_ = CHANNELS;
25     const int height_ = HEIGHT;
26     const int width_ = WIDTH;
27
28     //开辟CPU 内存
29     ...
30
31     //读取数据文件
32     ...
33
34     //初始化设备
35     ...
36
37     //选择 UNION 模式
38     switch (core_num_) {
39     ...
40     }
41
42     // float2half
43     ...
44
45     // Passing param
46     ...
47
48     // create cnrt Notifier
49     ...
50
51     // hardware time
52     cnrtPlaceNotifier(Notifier_start, pQueue);
53     // 启动kernel
54     CNRT_CHECK(cnrtInvokeKernel_V2(...));
55     cnrtPlaceNotifier(Notifier_end, pQueue);
56     CNRT_CHECK(cnrtSyncQueue(pQueue));
57
58     gettimeofday(&end, NULL);
59     float time_use = ((end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec))/1000.0;
60     printf("time use: %.3f ms\n", time_use);
61
62     float* output_tmp = (float*)malloc(data_count * sizeof(float));
63     ...
64 }
```

```
1 // plugin_sbc_op.cc
2 cnmlStatus_t cnmlCreatePluginSBCOpParam(
3     cnmlPluginSBCOpParam_t *param,
4     int batch_num_
5 ) {
6     *param = new cnmlPluginSBCOpParam();
7     (*param)->batch_num_ = batch_num_;
8
9     return CNML_STATUS_SUCCESS;
10 }
11
12 cnmlStatus_t cnmlDestroyPluginSBCOpParam(
13     cnmlPluginSBCOpParam_t *param
14 ) {
15     delete (*param);
16     *param = nullptr;
17
18     return CNML_STATUS_SUCCESS;
19 }
20
```

图 7.43 CNPlugin 添加头文件接口声明

```
1 // plugin_sbc_op.cc
2 cnmlStatus_t cnmlCreatePluginSBCOp(
3     ... // 参数定义
4 ) {
5
6     int input_num = 1;
7     int output_num = 1;
8
9     void** InterfacePtr;
10    InterfacePtr = reinterpret_cast<void**>(&SBCKernel);
11
12    cnrtKernelParamsBuffer_t params;
13    // 向params添加参数
14    ...
15
16    // 调用cnmlCreatePluginOp进行创建
17    cnmlCreatePluginOp(...);
18
19    cnrtDestroyKernelParamsBuffer(params);
20    return CNML_STATUS_SUCCESS;
21 }
22
23 cnmlStatus_t cnmlComputePluginSBCOpForward(
24     ... // 参数定义
25 ) {
26
27     cnrtInvokeFuncParam_t compute_forw_param;
28     // 参数初始化
29     ...
30     // 调用cnmlComputePluginOpForward
31     cnmlComputePluginOpForward_V3(...);
32
33     return CNML_STATUS_SUCCESS;
34 }
35
```

图 7.44 CNPlugin 算子注册

```

1 // math_ops.cc
2 REGISTER_OP("SBC")
3 .Input("input: T")
4 .Output("output: T")
5 .Attr("T:type")
6 .SetShapeFn(shape_inference::UnchangedShape);
7

```

图 7.45 EAST 算子注册

```

1 // mlu_lib_ops.cc
2 tensorflow::Status CreateSBCOp(MLUBaseOp** op, MLUTensor* input,
3                                 MLUTensor* output, int batch_num_) {
4
5     MLUTensor* inputs_ptr[1] = {input};
6     MLUTensor* outputs_ptr[1] = {output};
7
8     CNML_RETURN_STATUS(cnmlCreatePluginSBCOp(op, inputs_ptr, outputs_ptr, batch_num_));
9 }
10
11 tensorflow::Status ComputeSBCOp(
12     MLUBaseOp* op,
13     void* input,
14     void* output,
15     MLUCnrtQueue* queue) {
16
17     void* inputs_ptr[1] = {input};
18     void* outputs_ptr[1] = {output};
19
20     CNML_RETURN_STATUS(cnmlComputePluginSBCOpForward(
21                                     op, inputs_ptr, 1, outputs_ptr, 1, queue));
22 }
23
24 tensorflow::Status CreateSBCOp(MLUBaseOp** op, MLUTensor* input,
25                                 MLUTensor* output, int batch_num_);
26
27 tensorflow::Status ComputeSBCOp(
28     MLUBaseOp* op,
29     void* input1,
30     void* output,
31     MLUCnrtQueue* queue);
32

```

图 7.46 EAST MLULib 层

```

1 // mlu_ops.h
2 DECLARE_OP_CLASS(mlu...);
3 DECLARE_OP_CLASS(MLUSBC);
4 DECLARE_OP_CLASS(mlu...);
5

```

图 7.47 east-mluops1

```

1 // sbc.cc
2 /* Copyright 2018 Cambrian*/
3 #if CAMBRICON_MLU
4
5 namespace stream_executor {
6 namespace mlu {
7 namespace ops {
8
9 Status MLUSBC::CreateMLUOp(std::vector<MLUTensor *> &inputs,
10                           std::vector<MLUTensor *> &outputs, void *param) {
11   TF_PARAMS_CHECK(inputs.size() > 0, "Missing input");
12   TF_PARAMS_CHECK(outputs.size() > 0, "Missing output");
13
14   MLUBaseOp *op_ptr = nullptr;
15   MLUTensor *input = inputs.at(0);
16   MLUTensor *output = outputs.at(0);
17
18   int batch_num_ = *((int *)param);
19
20   MLULOG(3) << "CreateSBCOp"
21     << ", input: " << lib::MLUTensorUtil(input).DebugString()
22     << ", output: " << lib::MLUTensorUtil(output).DebugString();
23
24   // 调用接口进行创建
25   TF_STATUS_CHECK(lib::CreateSBCOp(...));
26
27   base_ops_.push_back(op_ptr);
28
29   return Status::OK();
30 }
31
32 Status MLUSBC::Compute(const std::vector<void *> &inputs,
33                        const std::vector<void *> &outputs, cnrtQueue_t queue) {
34   void *input = inputs.at(0);
35   void *output = outputs.at(0);
36
37   // 调用接口进行计算
38   TF_STATUS_CHECK(lib::ComputeSBCOp(...));
39
40   TF_CNRT_CHECK(cnrtSyncQueue(queue));
41
42   return Status::OK();
43 }
44 } // namespace ops
45 } // namespace mlu
46 } // namespace stream_executor
47
48#endif // CAMBRICON_MLU
49
50

```

图 7.48 east-mluops2

```

1 // mlu_stream.h
2 Status SBC(OpKernelContext* ctx, Tensor* input, Tensor* output, int batch_size) {
3   return CommonOpImpl<ops::MLUSBC>(...);
4 }
5

```

图 7.49 east-mlustream

```

1 // cwise_op_sbc_mlu.h
2 #ifndef TENSORFLOW_CORE_KERNELS_CWISE_OP_POWER_DIFFERENCE_MLU_H_
3 #define TENSORFLOW_CORE_KERNELS_CWISE_OP_POWER_DIFFERENCE_MLU_H_
4 #if CAMBRICON_MLU
5
6 namespace tensorflow {
7 template <typename T>
8 class MLUSBCOp : public MLUOpKernel {
9 public:
10   explicit MLUSBCOp(OpKernelConstruction* ctx) :
11     MLUOpKernel(ctx) {}
12
13   void ComputeOnMLU(OpKernelContext* ctx) override {
14
15     if (!ctx->ValidateInputsAreSameShape(this)) return;
16     auto* stream = ctx->op_device_context()->mlu_stream();
17     auto* mlustream_exec = ctx->op_device_context()->mlu_stream()->parent();
18     Tensor input = ctx->input(0);
19
20     // 参数检查与处理
21     .....
22
23     // 输出形状推断及输出内存分配
24     .....
25
26     // 调用MLUStream层接口完成算子计算
27     .....
28
29   }
30 };
31
32 } // namespace tensorflow
33
34 #endif // CAMBRICON_MLU
35 #endif // TENSORFLOW_CORE_KERNELS_CWISE_OP_SQUARED_DIFFERENCE_MLU_H_
36

```

图 7.50 east-MLUOpKernel

```

1 // cwise_op_sbc_mlu.cc
2 namespace tensorflow {
3 #if CAMBRICON_MLU
4 #define REGISTER_MLU(T)
5   REGISTER_KERNEL_BUILDER(Name("SBC") \
6     .Device(DEVICE_MLU) \
7     .TypeConstraint<T>("T"), \
8     MLUSBCOp<T>);
9 TF_CALL_MLU_FLOAT_TYPES(REGISTER_MLU);
10 #undef REGISTER_MLU
11 #endif
12 }
13

```

图 7.51 east-MLUOpkernel2

```
1 node {
2     name: "concat"
3     op: "SBC"
4     input: "input_images"
5     batch_num_: "1"
6     attr {
7         key: "T"
8         value {
9             type: DT_FLOAT
10        }
11    }
12 }
```

图 7.52 east-pbtxt1

```
1 node {
2     name: "split/split_dim"
3     op: "Const"
4     attr {
5         key: "dtype"
6         value {
7             type: DT_INT32
8         }
9     }
10    attr {
11        key: "value"
12        value {
13            tensor {
14                dtype: DT_INT32
15                tensor_shape {
16                    }
17                int_val: 3
18            }
19        }
20    }
21 }
22 .....
23 node {
24     name: "concat"
25     op: "ConcatV2"
26     input: "sub"
27     input: "sub_1"
28     input: "sub_2"
29     input: "concat/axis"
30     attr {
31         key: "N"
32         value {
33             i: 3
34         }
35     }
36     attr {
37         key: "T"
38         value {
39             type: DT_FLOAT
40         }
41     }
42     attr {
43         key: "Tidx"
44         value {
45             type: DT_INT32
46         }
47     }
48 }
```

图 7.53 east-pbtxt2

7.3 自然语言处理-BERT

7.3.1 实验目的

本实验主要完成将自然语言处理的代表性算法——BERT 网络移植到智能处理器 DLP 上，并进行性能优化与比较，使读者可以借助 DLP 处理器完成完整的自然语言处理任务。

因此，本实验的主要目的包括：

1. 通过用智能编程语言实现 BERT 模型中的 BatchMatMulV2 算子，深入掌握智能编程语言的算子开发和优化方法；
2. 通过在 TensorFlow 中添加大算子，掌握在 TensorFlow 框架中添加融合算子的方法；
3. 通过使用 TensorFlow 进行在线推理，掌握使用 TensorFlow 编写目标检测应用并在典型 DLP 上进行优化的方法；
4. 通过与 BANGC 大算子实现的比较，体会 DLP 相比 GPU 和 CPU 的优势。

实验时间预计为两周。

7.3.2 背景介绍

7.3.2.1 NLP 介绍

总的来说，NLP 的目标是要让计算机正确处理人类的自然语言。常见的 NLP 问题一般包括阅读理解，问答系统，对话系统，文本摘要和文本分类等。与一般的分类或者检测网络不同，NLP 相关的深度学习技术一般都关注于字符之间的序列关系。

本实验依赖的数据集为斯坦福问答数据集(Stanford Question Answering Dataset, SQuAD)是一种阅读理解数据集，由工作人员在一组 Wikipedia 文章上提出的问题组成，其中每个问题的答案是对应阅读文章或问题的一段文本，而问题本身可能是无法回答的。

例如对于文本：

James Bryant Conant led the university through the Great Depression and World War II and began to reform the curriculum and liberalize admissions after the war.

有对应问题：

What was the name of the leader through the Great Depression and World War II?

算法应当正确回答：

James Bryant Conant

更详细的内容请参考数据集的官方链接：<https://rajpurkar.github.io/SQuAD-explorer/>

7.3.2.2 NLP 常见算法、原理

1. **LSTM** 关于 LSTM 的相关原理，在智能计算系统理论书中已经有所介绍。其是在循环神经网络(RNN)的基础上，将 RNN 的隐层单元用 LSTM(Long Short Term Memory) 单元来代替，使其能保留更长的时间信息，减缓梯度消失的现象。相较于卷积神经网络，循环神经网络有存储信息的能力，可有效处理序列数据。此外由于其权值共享的特性，循环神经网络的参数一般要少于卷积神经网络，其结构图如下图 7.54 所示。

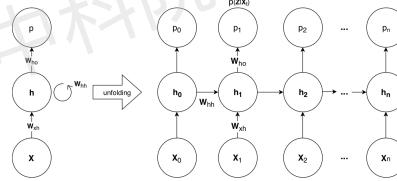


图 7.54 循环神经网络结构图

2. **Attention** Attention 是一种用于提升基于 RNN (LSTM 或 GRU) 的 Encoder + Decoder 模型效果的机制，一般也称为注意力机制。注意力机制现已广泛应用于机器翻译、语音识别、图像标注 (Image Caption) 等领域。Attention 之所以受欢迎的原因在于其给模型赋予了区分辨别的能力，例如，在机器翻译、语音识别应用中，为句子中的每个词赋予不同的权重，使神经网络模型的学习变得更加灵活 (soft)。具体来说，注意力机制可以帮助模型对输入的每个部分赋予不同的权重，抽取出更加关键及重要的信息，使模型做出更加准确的判断，同时不会对模型的计算和存储带来更大的开销。

对于一般的 Seq2Seq 模型，其通常在编码阶段 (Encoder) 把输入编码成一个固定的长度，这样做会带来两个问题：当输入序列很长时，模型性能会有影响；其对输入的每个元素赋予相同的权重，无法体现重要程度。而注意力机制一定程度上可以解决此问题，其会对输入的每一个元素计算出其对输出端各个元素之间的权重，表示源端第 i 个元素对目标端各元素的影响程度。

常见的注意力机制可分为许多种类，常见的有 soft Attention 和 Hard Attention；Global Attention 和 Local Attention；以及 Self Attention 等。其中 Self Attention 为 Transformer 的重要部分，此处简单对齐作出介绍。

Self Attention 与传统的 Attention 机制不同：传统的 Attention 如上所述是基于输入端和输出端的隐层单元计算 Attention 的，得到的结果是源端的每个词与目标端每个词之间的依赖关系。而 Self Attention 是分别对输入端与输出端自身进行计算，捕捉输入端或输出端自身的词与词之间的依赖关系；然后再把输入端得到的 self Attention 加入到输出端得到的 Attention 中，捕捉输入端和输出端词与词之间的依赖关系。因此 Self Attention 最终得到的信息中不仅包含输入与输出的依赖关系，还包含自身词与词之间的依赖关系，其效果要好于传统的注意力机制。

3. **Transformer** 如图 7.55 所示，其编码部分输入 (*inputs*) 对应训练的数据集，解码部分输入 (*outputs-shifted right*) 对应数据集的标签。

像大多数 Seq2Seq 模型一样，Transformer 也分为 Encoder 和 Decoder 两个部分组成（分别对应图 7.55 中的左半部分和右半部分）。

对于 Encoder 来说，其由 6 个相同的 Layer 组成 (即 $N=6$)，每个 Layer 又由两个 Sub-Layer 组成，分别是 multi-head self-attention mechanism 和 fully connected feed-forward network。其中每个 Sub-Layer 还增加了 Residual Connection 和 Layer Normalisation。

对于 Decoder 来说，其在 Encoder 的基础上，增加了额外的 Attention Sub-Layer。该子层同时增加了 Mask 操作，以此确保预测第 i 个位置时仅可以获取到 i 之前的输出（不会接触到未来的信息）。在训练时其输入为 Encoder 的输出，以及编码器输入对应的标签（整

体向右偏移一位)，由于标签是始终存在的，因此训练过程是可以做到并行的。在推理时，Decoder 没有 ground truth 作为输入，其输入来自上一个位置的输出，也就意味着需要逐个位置进行解码，无法并行。

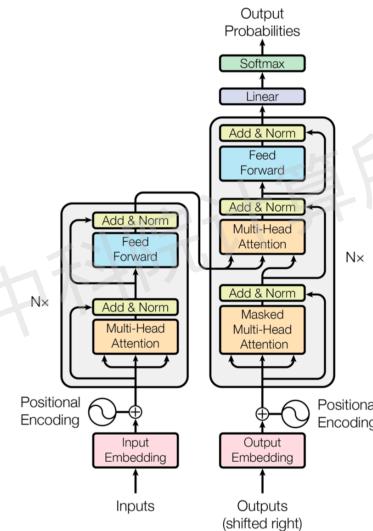


图 7.55 Transformer 结构图 (1)

从图 7.55 和图 7.56 的结构示意图可以发现，Encoder 和 Decoder 是层叠多个 Multi-Head Attention 单元构成，而每一个 Multi-Head Attention 单元由多个结构相似的 Scaled Dot-Product Attention 单元组成。Self Attention 也是在 Scaled Dot-Product Attention 单元里面实现的，如上图左图所示，首先把输入经过不同的线性变换分别得到 Q、K、V (Q、K、V 都来自于相同的输入)。然后把 Q 和 K 做 dot Product 相乘，得到输入词与词之间的依赖关系，然后经过尺度变换 (scale)、掩码 (Mask) 和 Softmax 操作，得到最终的 Self Attention 矩阵。方程式表达如下，其中 key(K) 意味着原始隐层单元的输出，value(V) 代表每一个隐层单元之间的关系，Q 为 outputs-shifted right 部分的输出。

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

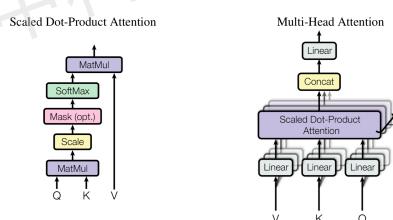


图 7.56 Transformer 结构图 (2)

4. BERT BERT 的全称为 Bidirectional Encoder Representation from Transformers。其采用了 Transformer 的 Encoder 部分，并且进行双向的 Block 连接，与 Transformer 相比，不仅

可获取到语句之前的信息，还可获取到未来的信息。其结构图如图 7.57 所示。

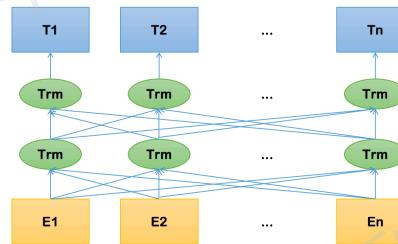


图 7.57 BERT 结构图

图中的黄色部分为输入的嵌入（Embedding）表示，其由三种不同的 Embedding 求和而成，分别为 Token Embeddings，Segment Embeddings 以及 Position Embeddings。Token Embeddings 是模型输入中每个元素的词向量表示；Segment Embeddings 用来区分输入的前后两个语句；Position Embeddings 含义与 Transformer 相同，但 Transformer 中使用不同频率的三角函数直接计算得出，而 BERT 是通过训练学习得到的。图 7.57 中每一个黄色框均代表模型输入的一个元素对应的 Embedding 表示。紧接着将 Embedding 输入到 Transformer 的编码（Encoder）部分，由于 Encoder 没有对 Multi-Head Attention 进行 Mask 操作，这也是 BERT 能获取到双向信息的原因。经过多层串联的 Transformer 后即得到最终的输出（图 7.57 中的蓝色框部分）。

在训练时，BERT 包含两个预训练子任务。第一个子任务为 Masked Language Model，通过随机遮挡每一个句子中 15% 的词，用其上下文来做预测，在计算损失函数时只计算被遮挡掉的输入元素。第二个子任务为 Next Sentence Prediction，可以理解简单的分类任务，用以判断两个输入语句是否为前后关系。

使用预训练模型进行微调时，BERT 针对不同的预测任务给予了不同的方法。如下图 7.58 所示，由此可以看出 BERT 不仅适合做简单的文本分类、序列标注等常见任务，对于问答系统、信息检索、聊天机器人等任务都有着重大的突破。在本实验中，我们完成的是问答系统的任务，对应与图 7.58 中的（c）部分。

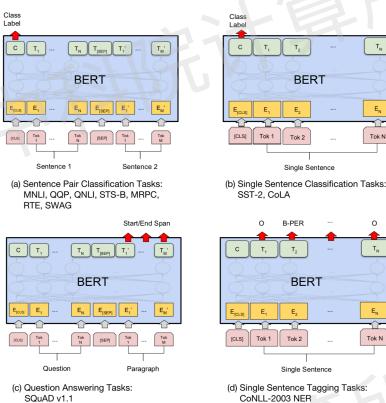


图 7.58 BERT Fine Tuning

7.3.2.3 BatchMatMul 算子

BatchMatMulV2 算子是 BERT 模型中重要的一个操作，其主要进行两个张量相乘的操作，其中这两个输入张量的 shape 可能不一致，因此还涉及到 BroadCast 操作，其还有两个参数可控制两个输入张量进行求解伴随矩阵操作。在本实验中，出于简化实验难度的考虑，在此处将求解伴随矩阵操作简化为普通的转置操作 (transpose)。

BatchMatMulV2 的输入输出如下：

Arguments: x: 2 维或更高维度，输入 shape 为 [...,r_x,c_x] y: 2 维或更高维度，输入 shape 为 [...,r_y,c_y]

Attrx: adj_x: 当为 True 时，求解输入 x 的转置矩阵，默认为 False adj_y: 当为 True 时，求解输入 y 的转置矩阵，默认为 False

Return: Output: 返回 3 维或更高维度的张量，输出 shape 为 [...,r_o,c_o]

BatchMatMul 和普通矩阵乘的区别在于 BatchMatMul 在 Batch 维度上对输入张量进行拆分，并将拆分后的张量分别执行矩阵乘操作，最后再将矩阵乘的结果在 Batch 维度上拼合。BatchMatMulV2 其和 BatchMatMul 算子的区别在于 BatchMatMulV2 支持 Batch 维度的 broadcasting 操作。

7.3.3 实验环境

本实验所涉及的硬件平台和软件环境与其它综合实验相同。

7.3.4 实验内容

1. BERT 重训练 (可选): 通过 Fine-Tuning 的方式，进一步提升模型的精度。目前 DLP 还不支持训练，学生可以使用 CPU 或 GPU 完成模型的重训练工作，也可直接使用教材提供的模型做下一步实验。
2. BERT DLP 移植: 将 BERT 网络移植在 DLP 中，包含模型量化，config 设置等操作。
3. BANGC BatchMatMulV2 算子实现与集成。
4. 分别比较 CPU 性能、MLU 性能 (BatchMatMulV2 放在 CPU 上计算)、MLU 性能 (BatchMatMulV2 使用 BANGC 实现)、BERT 大算子网络性能。

如图7.59中虚线框出的文件是本实验需要修改的文件内容。其中主要包括了 BatchMatMulV2 的 BCL 实现和 TensorFlow 集成。

7.3.5 实验步骤

7.3.5.1 BERT 重训练

由于 BERT 只提供了预训练模型，在做特点任务时，需要先做微调训练。

首先在 GitHub 中，拉取工程：<https://github.com/google-research/bert.git>

根据提示，分别下载模型和训练测试数据集。在这里，我们选择 SQuAD 1.1 任务，因此需要分别下载 Bert-Base(Uncased, 12-layer, 768-hidden, 12-heads, 110M parameters) 模型以及 train-v1.1.json、dev-v1.1.json 数据集。

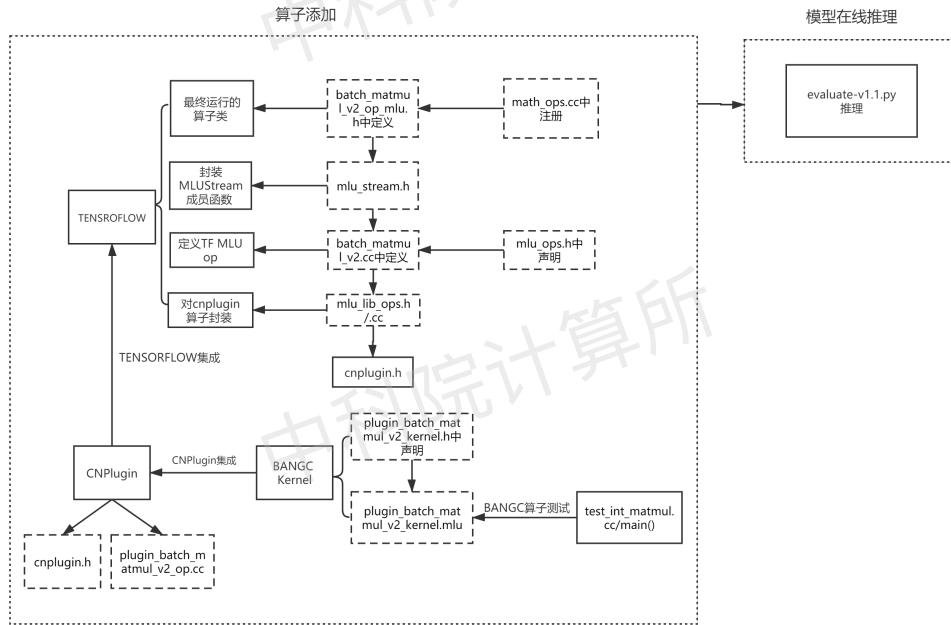


图 7.59 实验内容

执行如下所示的命令，在 CPU 或 GPU 环境中进行训练，并验证精度。

```

1 export BERT_BASE_DIR=/path/to/bert-model/uncased_L-12_H-768_A-12
2 export SQUAD_DIR=/path/to/dataset/squad
3 python run_squad.py \
4   --vocab_file=$BERT_BASE_DIR/vocab.txt \
5   --bert_config_file=$BERT_BASE_DIR/bert_config.json \
6   --init_checkpoint=$BERT_BASE_DIR/bert_model.ckpt \
7   --do_train=True \
8   --train_file=$SQUAD_DIR/train-v1.1.json \
9   --do_predict=True \
10  --predict_file=$SQUAD_DIR/dev-v1.1.json \
11  --train_batch_size=12 \
12  --learning_rate=3e-5 \
13  --num_train_epochs=2.0 \
14  --max_seq_length=384 \
15  --doc_stride=128 \
16  --output_dir=/path/to/squad_base/
17
18 python $SQUAD_DIR/evaluate-v1.1.py $SQUAD_DIR/dev-v1.1.json /path/to/squad_base/
  predictions.json

```

图 7.60 BERT Fine Tuning

执行完上述指令后，会打印相应的输出精度：“f1”: xx.xx, “exact_match”: xx.xx。

由于微调训练实验耗时较长且对硬件环境有所要求，不作为本章的重点，学生只需了解训练的流程，在实际实验阶段可直接使用 DLP 提供的训练模型进行推理实验。

7.3.5.2 BERT DLP 移植

为了将 BERT 网络推理在 DLP 硬件上，需要对相关代码进行修改移植。与之前的综合实验类似，首先需要增加 DLP 专属的 config 配置，其次还需要对模型进行量化操作。为了方便推理，还在原有的 run_squad.py 文件中增加了 checkpoint 模型转 pb 模型的操作，以此简化后续调试的流程。该部分的代码修改增加部分如下图 7.61 所示：

从上图中代码可知，在将 checkpoint 模型转 pb 模型时，由于 checkpoint 格式不包含模型的结构信息，所以需要首先在 session 实例中构造图的结构，然后加载相应的模型并作初始化。紧接着需要将计算图中的变量 (variable) 转为常量 (constant)，指定输出节点并做序列化为字符的操作。最终生成的 pb 模型相比原模型仅保留了与输出节点相关的节点，相当于在原模型的基础上做了剪枝的操作，并且保存的常量在推理时不需要初始化操作，进一步提高了推理时的性能。

在配置 DLP 相关的 config 时，增加了 optype_black_list 设置，经过该设置会将配置的节点自动执行在 CPU 而非 DLP 设备上。可以将一些 DLP 支持但由于性能精度暂不满足要求的算子暂时执行在 CPU 上。

在进行量化时，为了兼顾性能与精度的要求，本次实验选择 INT16 作为量化精度。首先进行配置文件 bert_int16.ini 的编写，然后执行 DLP 提供的量化脚本 fppb_to_intpb.py 得到量化后的模型。在该实验中，DLP 会将 MatMul、MLP、BatchMatMulV2 等操作量化为 INT16 精度，其它节点仍然以 FP32 或 FP16 的精度进行运算。该部分代码如图 7.62 所示：

7.3.5.3 BatchMatMulV2 BANGC 代码实现

完成 DLP 移植后，推理时会发现 DLP 暂时不支持 BatchMatMulV2 算子，由于 BERT 中存在大量的 BatchMatMulV2 操作，不仅会导致 CPU 利用率的升高，还会由于大量的分段造成性能的下降。为了解决该问题，我们设计了 BatchMatMulV2 算子的 BCL 实现，在减小分段的同时通过 INT16 定点计算减小时延提高效率。

1. 设计目标使用 BCL 实现 BatchMatMulV2 算子，并以 cnplugin 封装，供框架直接调用。分别在 plugin_batch_matmul_v2_kernel.h 和 plugin_batch_matmul_v2_kernel.mlu 进行算子的定义和实现。此模块的参数规格设计为：

输入矩阵 1：input_0[dim_0, dim_1, m, k], fp16；输入矩阵 2：input_1[dim_0, dim_1, n, k], fp16；输出矩阵：output[dim_0, dim_1, m, n], fp16

2. 设计思路考虑到使用循环向量乘指令不能很好地体现 MLU 的矩阵运算性能，因此设计使用卷积指令来执行 MatMul，右矩阵将会在量化完成后拷贝至 wram 空间，使用 __bang_conv 指令进行计算。因为最终是要调用卷积指令来进行矩阵乘法，所以代码中 n, ci, co 是沿用的卷积指令符号系统，这与接口中 m, n, k 的对应关系是：m → n_；n → co_；k → ci_。其中 n_ × ci_ 为卷积的输入，ci_ × co_ 为卷积的 filter，n_ × co_ 为卷积的输出。

3. 函数设计

- **_mlu_entry_ void BatchMatMulV2Kernel_MLU270_half ()**

入口函数，主要用于开辟 NRAM、WRAM 和 SRAM 的空间，遍历 dim_0 和 dim_1 循环进行卷积，并调用 int_matmul_wrap 函数进行计算，其主要参数如下：

void* left_ddr: GDRAM 上的左矩阵
 void* right_ddr: GDRAM 上的右矩阵
 void* dst_ddr: GDRAM 上的输出矩阵
 int dim_0: 输入维度, 左右矩阵的第一维
 int dim_1: 输入维度, 左右矩阵的第二维
 int m: 输入维度, 左矩阵的第三维
 int n: 输入维度, 右矩阵的第四维
 int k: 输入维度, 左矩阵的第四维, 右矩阵的第三维
 float scale_0: 左矩阵的量化 scale 参数
 int pos_0: 左矩阵的量化 pos 参数
 float scale_1: 右矩阵的量化 scale 参数
 int pos_1: 右矩阵的量化 pos 参数

- **template <typename IT, typename FT>**

- _mlu_func_ void int_matmul_wrap():**

主要计算流程函数, 将左右矩阵分别量化并拷贝至对应的空间, 调用 compute 函数进行计算, 然后将结果拷贝回 GDRAM 上, 其主要参数如下(省略与上文重复的参数, 下同):

IT *nram_buf: NRAM 上开辟的用于计算的空间

IT *wram_buf: WRAM 上开辟的用于计算的空间

IT *sram_buf: SRAM 上开辟的用于计算的空间

- **template <typename T> _mlu_func_ void load_left():**

将左矩阵拷贝至 NRAM 上。并进行量化操作, 然后拷贝至 SRAM 上, 将 NRAM 的空间空余出来用于量化和摆放右矩阵。

- _mlu_func_ void SegStrategyByFactor():**

m, n 拆分策略函数。按照预设的 factor 搜索 n 值, 并按照此 n 值拆分 m 值。

int factor: 预设的 n 的拆分倍数

int k_up: k 维度进行 64 位对齐的结果

int byteit: 输入数据类型的字节数

int byteft: 输出数据类型的字节数

int &work_m: (输出) 拆分后每次读取的 m 值

int &work_n: (输出) 拆分后每次读取的 n 值

- * _mlu_func_ void SegStrategyByMaxCo():**

最大空间的 m, n 拆分策略函数, 按照最大 WRAM 空间寻找 n 值, 并按照此 n 值拆分 m 值。

- template <typename T> _mlu_func_ void load_right():**

将右矩阵拷贝至 NRAM 上。并进行量化操作, 然后拷贝至 WRAM 上。

T *right_inchip: WRAM 上的最大空间

T *right_ddr: GDRAM 上的右矩阵

T *nram_buf: 临时 NRAM 空间, 用于量化右矩阵

int n_in_nram: 根据 k_up 计算出的 NRAM 上的剩余空间一次所能计算的最大 n 值

- **template <typename IT, typename FT> __mlu_func__ void compute():**

计算卷积，得到 MatMul 的结果

FT *dst: 存放结果的 NRAM 空间

IT *left: 存放在 NRAM 上的左矩阵

IT *right: 存放在 WRAM 上的右矩阵

FT recip_scale: 量化 scale 参数

int pos: 量化 pos 参数

4. 具体实现

- 准备所需的变量，进行空间划分。
- 最外层对 n 方向和 c 方向使用两重循环，保证多维度输入结果的准确性。
- 按照最大 SRAM 空间进行 m 维度拆分: $\text{work_m} = \text{SRAM_BUF_SIZE} / \text{sizeof}(\text{TYPE}) / k$

记为 work_m，循环调用 int_matmul_wrap 计算维度为 [work_m, n, k] 的 MatMul。如下图所示：图中蓝色阴影部分即为 SRAM_BUF_SIZE

- 在每个 int_matmul_wrap 运算中， $\text{work_m} * k < \text{SRAM_BUF_SIZE}$ ，所以能保证完全载入到 SRAM 空间上，故先执行 load_left，将 input_0 数据 load 到 SRAM 空间上。
- 调用 SegStrategyByFactor 和 SegStrategyByMaxCo 函数寻找 n 和 work_m 的拆分策略，其中一种拆分策略为：

```
int work_n_1 = WRAM_BUF_SIZE / byteit / k_up;
int work_n_2 = (NRAM_BUF_SIZE - byteit * k_up) / byteft;
work_n = ALIGN_DN(MIN(work_n_1, work_n_2), 64);
int work_m_1 = NRAM_BUF_SIZE / (byteit * k_up + byteft * work_n);
int work_m_2 = (NRAM_BUF_SIZE - byteit * 64 * k_up) / (byteit * k_up);
work_m' = MIN(work_m_1, work_m_2);
```

将上述的计算结果记为 work_n, work_m'。

通过循环 `for (int cur_n = 0; cur_n < n; cur_n += taskDim * work_n)` 计算维度为 [work_m, work_n, k] 的 MatMul，流程图如 7.65 所示：

图中黄色阴影部分即为 $\text{work_n} * \text{k_up} * \text{byteit}$ ，根据选取的拆分策略不同可能为 WRAM_BUF_SIZE 也可能为较小值。

- 在每个 [work_m, work_n, k] 的 MatMul 计算中，因为拆分策略中已经保证 $\text{work_n} * \text{k_up} * \text{byteit} < \text{WRAM_BUF_SIZE}$ ，所以能保证完全载入到 WRAM 空间上，故执行 load_right，将 input_1 数据 load 到 WRAM 空间上。

- 在每个 [work_m, work_n, k] 的 MatMul 计算中，还需要按照 work_m' 进行拆分，以保证左矩阵和输出矩阵有足够的空间放在 NRAM 上，如下图 7.66 所示：

上图中将 output 展示出来是由于 output 的结果与 input_0 共用 NRAM 空间，拆分策略中已经保证 $(\text{work_n} * \text{byteit} + \text{k_up} * \text{byteft}) * \text{work_m}' < \text{NRAM_BUF_SIZE}$ ，所以输入输出均能完整的载入到 NRAM 空间上。

- 进行 compute 计算，计算每个 [work_m', work_n, k] 的 MatMul。
- 多重循环中按照输出结果应有的地址偏移将 output 拷出到 GDRAM 上。

7.3.5.4 BatchMatMulV2 BANGC 代码集成

类似于前两个实验的算子添加过程，集成主要分为 CNPlugin 的集成和 TensorFlow 框架的集成。

1. CNPlugin 集成：分别完成

```
cnmlCreatePluginBatchMatMulV2OpParam();
cnmlDestroyPluginBatchMatMulV2OpParam();
cnmlCreatePluginBatchMatMulV2Op()
```

以及 `cnmlStatus_t cnmlComputePluginBatchMatMulV2OpForward()` 构造函数。

传入 BCL 计算程序所需的参数： $scale_0, pos_0, scale_1, pos_1$ (代表算子两个输入的量化参数)； dim_0, dim_1 (代表两个输入的第 0 个和第 1 个维度的值)； m, n, k (代表第一个输入的最后两个维度为 [m,k]，第一个输入的最后两个维度为 [n,k])。

此外还需在 `cnplugin.h` 头文件中增加相应的接口，如下图 7.68 所示：

2. TensorFlow 集成：

- 算子注册，如 7.68 所示：

• 定义 MLULib 层接口，用以连接 CNPlugin 的封装接口。这部分代码比较直接，如 7.69 所示；

• 定义 MLUOp 层接口，实现算子类的 `Create` 和 `Compute` 方法。要注意的是在这一步可以实现上面没有实现的转置操作。具体原理为借助 CNML 已经实现的转置算子通过底层算子拼接的方式来实现高层的算子功能。这里的代码当判断要求对输入转置的时候，会将首先调用 `lib::CreateTransposeProOp` 完成数据转置，再将输出结果作为 `lib::CreateBatchMatMulV2Op` 的输入。需要注意的是，这种算子拼接在 `Create` 和 `Compute` 部分都需要执行，如 7.70 所示。

- 定义 MLUStream 层接口，如 7.71 所示，这部分也比较简单，不再赘述。

• 定义 MLUOpKernel 层接口：在 `tensorflow/core/kernels/batch_matmul_v2_op_mlu.h` 文件中完成以下函数，并调用 MLUStream 层的 `BatchMatMulV2` 函数，如 7.72 所示。

分别完成 CNPlugin 和 TF 的集成后，即可运行正常的推理代码，完成 SQuAD 1.1 验证数据集的测试。完整的推理执行命令如图 7.73

7.3.5.5 BERT 大算子

为了进一步比较 BCL 的优越性，DLP 还实现了完整 BERT BCL 大算子的推理。其在 pb 模型中仅使用一个 CNPlugin 中的集成算子代替其它所有小算子，因此避免了多余的分段。同时由于开发者可以通过 BCL 自由控制硬件计算资源和内存的使用，进一步提高了片上内存的使用效率，极大的提升了整体端到端的性能。

使用者在运行 BERT 大算子模型时，仍然使用 DLP 移植后的开源代码，仅需将模型来源指定为相应的 pb 模型。

在生成大算子模型时，与第 7.3.5.2 章节不同的是，其在量化配置文件中选择大算子作为输出节点。配置文件如下图所示：

运行推理代码后，会发现此种方式下性能有着大幅度的提升。

7.3.6 实验评估

相比其它两个综合实验，BERT 实验困难度最高，因此难度系数分为 1.2。学生最终得分为 1.2 乘以相应的得分。

- 60 分标准：完成 BatchMatMulV2 BCL 算子的实现与 cnplugin 的集成，可以跑通单算子测试程序。
- 70 分标准：在 60 分的基础上，执行单算子测试脚本时，错误率在 1% 以内。
- 80 分标准：在 70 分的基础上，完成 TensorFlow 算子集成，执行完整的 BERT 模型推理验证脚本，f1 值大于 80、exact_math 值大于 70，单 Batch 单次推理平均延时小于 100ms。
- 90 分标准：在 80 分的基础上，执行测试脚本时，f1 值大于 82、exact_math 值大于 73，单 Batch 单次推理平均延时小于 60ms。
- 100 分标准：在 90 分的基础上，执行测试脚本时，f1 值大于 82.5、exact_match 值大于 74，单 Batch 单次推理平均延时小于 50ms。

7.3.7 实验思考

1. 本章提供的样例代码中通过使用卷积来做矩阵运算，可否使用矩阵乘等基本 BCL 操作来完成？
2. 使用上述不同方式完成 BCL 算子实现有何不同？哪种效率更高？
3. BatchMatMulV2 BCL 算子的两个输入最后两个维度为何为 [m,k]、[n,k] 形式？而不是 [m,k]、[k,n] 形式，这对 TensorFlow 框架集成 BCL 算子时调用 CNML 有何影响。
4. 与大算子 BERT 相比，有何方法可以进一步提升小算子 BERT 网络的性能？

```

1 // run_squad.py
2 from tensorflow.core.framework import graph_pb2
3 from tensorflow.python.framework import importer
4 from tensorflow.python.framework import ops
5 ...
6 flags.DEFINE_bool("export_frozen_graph", False, "Whether to export SavedModel.")
7 ...
8 def save_dlp_frozen_graph(bert_config, seq_len, init_checkpoint):
9     tf_config = tf.ConfigProto()
10    with tf.Session(config=tf_config) as tf_sess:
11        input_ids = tf.placeholder(tf.int32, [None, seq_len], 'input_ids')
12        input_mask = tf.placeholder(tf.int32, [None, seq_len], 'input_mask')
13        segment_ids = tf.placeholder(tf.int32, [None, seq_len], 'segment_ids')
14
15        # construct origin graph
16        (start_logits, end_logits) = create_model(
17            bert_config=bert_config,
18            is_training=False,
19            input_ids=input_ids,
20            input_mask=input_mask,
21            segment_ids=segment_ids,
22            use_one_hot_embeddings=False)
23
24        output_node_names = ['unstack']
25        tvars = tf.trainable_variables()
26        initialized_variable_names = {}
27        if init_checkpoint:
28            (assignment_map, initialized_variable_names
29             ) = modeling.get_assignment_map_from_checkpoint(tvars, init_checkpoint)
30
31        tf.train.init_from_checkpoint(init_checkpoint, assignment_map)
32        tf.logging.info("**** Load Variables Done ****")
33        tf_sess.run(tf.global_variables_initializer())
34    else:
35        raise ValueError("No init_checkpoint!")
36
37    tf.logging.info("**** Trainable Variables ****")
38    for var in tvars:
39        init_string = ""
40        if var.name in initialized_variable_names:
41            init_string = ", *INIT_FROM_CKPT*"
42        tf.logging.info(" name = %s, shape = %s%s", var.name, var.shape, init_string)
43
44    frozen_graph = tf.graph_util.convert_variables_to_constants(tf_sess,
45                    tf_sess.graph.as_graph_def(), output_node_names)
46
47    export_file = os.path.join(FLAGS.output_dir, "frozen_model.pb")
48    with tf.gfile.GFile(export_file, "wb") as f:
49        f.write(frozen_graph.SerializeToString())
50    tf.logging.info("**** Save Frozen Graph Done ****")
51 ...
52 def main(_):
53 ...
54     config = tf.ConfigProto(allow_soft_placement=True,
55                           inter_op_parallelism_threads=1,
56                           intra_op_parallelism_threads=1,
57                           log_device_placement=False)
58     config.mlu_options.core_version = "MLU270"
59     config.mlu_options.precision = "int16"
60     config.mlu_options.save_offline_model = False
61     config.mlu_options.core_num = 16
62     config.mlu_options.optype_black_list = "OneHot"
63     run_config = tf.contrib.tpu.RunConfig(
64         session_config=config)

```

图 7.61 BERT DLP 移植

```
1 // bert_int16.ini
2 [config]
3 quantization_type = int16
4 device_mode = clean
5 int_op_list = FC
6
7 [model]
8 activation_quantization_alg = naive
9 input_tensor_names = input_ids:0, input_mask:0, segment_ids:0
10 weight_quantization_alg = naive
11 seq_length = 128
12 original_models_path = /path/to/frozen/pb
13 output_tensor_names = unstack:0,unstack:1
14 save_model_path = /path/to/output/frozen_model_int16.pb
15
16 [data]
17 do_lower_case = False
18 doc_stride = 128
19 max_query_length = 64
20 batch_size = 8
21 num_runs = 1
22 vocab_file = bert/uncased_L-12_H-768_A-12/vocab.txt
23 data_list = bert/squad/dev-v1.1.json
```

图 7.62 BERT 量化

```
1 for (int cur_dim0 = 0; cur_dim0 < dim_0; cur_dim0++) {
2     for (int cur_dim1 = 0; cur_dim1 < dim_1; cur_dim1++) {
3         .....
4     }
5 }
```

图 7.63 两重循环

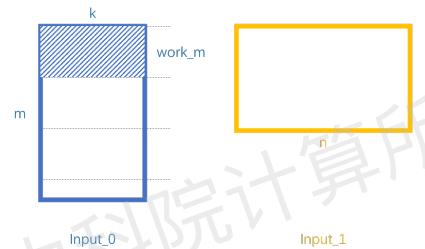


图 7.64 INPUT0 拆分

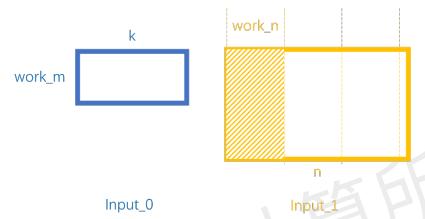


图 7.65 INPUT1 拆分

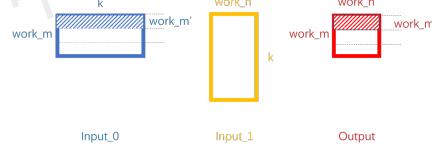


图 7.66 NRAM 拆分

```

1 // cnplugin.h
2 struct cnmlPluginBatchMatMulV2OpParam {...};
3
4 typedef cnmlPluginBatchMatMulV2OpParam *cnmlPluginBatchMatMulV2OpParam_t;
5
6 cnmlStatus_t cnmlCreatePluginBatchMatMulV2OpParam(...);
7
8 cnmlStatus_t cnmlDestroyPluginBatchMatMulV2OpParam(...);
9
10 cnmlStatus_t cnmlCreatePluginBatchMatMulV2Op(...);
11
12 cnmlStatus_t cnmlComputePluginBatchMatMulV2OpForward(...);

```

图 7.67 BERT-CNPlugin 接口

```

1 // tensorflow/core/kernels/batch_matmul_op_impl.h
2 #if CAMBRICON_MLU
3 #define REGISTER_BATCH_MATMUL_MLU(T) \
4     REGISTER_KERNEL_BUILDER( \
5         Name("MLUBatchMatMulV2") .Device(DEVICE_MLU) .TypeConstraint<T>("T") , \
6         MLUBatchMatMulV2<T>)
7 #endif // CAMBRICON_MLU
8
9 // tensorflow/core/ops/math_ops.cc
10 REGISTER_OP("MLUBatchMatMulV2")
11     .Input("x: T")
12     .Input("y: T")
13     .Output("output: T")
14     .Attr("T: {half, float}")
15     .Attr("input1_position: int = 0")
16     .Attr("input1_scale: float = 1.0")
17     .Attr("input2_position: int = 0")
18     .Attr("input2_scale: float = 1.0")
19     .Attr("adj_x: bool = false")
20     .Attr("adj_y: bool = false")
21     .SetShapeFn(shape_inference::BatchMatMulV2Shape)
22     .Doc(R"doc(
23 MLUBatchMatMulV2 (DLP only)
24 )doc");

```

图 7.68 算子注册

```
1 // tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.cc
2 tensorflow::Status CreateBatchMatMulV2Op(MLUBaseOp** op,
3                                         cnmlPluginBatchMatMulV2OpParam_t param,
4                                         MLUTensor* input1, MLUTensor* input2,
5                                         MLUTensor* output) {
6     MLUTensor* inputs[2];
7     MLUTensor* outputs[1];
8     inputs[0] = input1;
9     inputs[1] = input2;
10    outputs[0] = output;
11    CNML_RETURN_STATUS(cnmlCreatePluginBatchMatMulV2Op(op, param, inputs, outputs));
12 }
13
14 tensorflow::Status ComputeBatchMatMulV2Op(MLUBaseOp* op,
15                                             MLUCnrtQueue* queue,
16                                             void** inputs, int input_num,
17                                             void** outputs, int output_num) {
18     CNML_RETURN_STATUS(cnmlComputePluginBatchMatMulV2OpForward(
19         op, inputs, input_num, outputs, output_num, queue));
20 }
21
22 // tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.h
23 tensorflow::Status CreateBatchMatMulV2Op(MLUBaseOp** op,
24                                         cnmlPluginBatchMatMulV2OpParam_t param,
25                                         MLUTensor* input1, MLUTensor* input2,
26                                         MLUTensor* output);
27 tensorflow::Status ComputeBatchMatMulV2Op(MLUBaseOp* op,
28                                             MLUCnrtQueue* queue,
29                                             void** inputs, int inputs_num,
30                                             void** outputs, int outputs_num);
31
32 // tensorflow/stream_executor/mlu/mlu_api/ops/mlu_ops.h
33 struct MLUBatchMatMulV2OpParam {
34     float scale_0_;
35     int pos_0_;
36     float scale_1_;
37     int pos_1_;
38     int dim_0_;
39     int dim_1_;
40     int m_;
41     int n_;
42     int k_;
43     MLUBatchMatMulV2OpParam(float scale_0, int pos_0,
44                             float scale_1, int pos_1,
45                             int dim_0, int dim_1,
46                             int m, int n, int k)
47         : scale_0_(scale_0), pos_0_(pos_0), scale_1_(scale_1), pos_1_(pos_1),
48           dim_0_(dim_0), dim_1_(dim_1), m_(m), n_(n), k_(k) {}
49 };
50 ...
51 DECLARE_OP_CLASS(MLUBatchMatMulV2);
```

图 7.69 MLULib 接口

```

1 // tensorflow/stream_executor/mlu/mlu_api/ops/batch_matmul_v2.cc
2 #include "tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.h"
3 #include "tensorflow/stream_executor/mlu/mlu_api/ops/mlu_ops.h"
4 #include "tensorflow/stream_executor/mlu/mlu_api/tf_mlu_intf.h"
5 namespace stream_executor {
6 namespace mlu {
7 namespace ops {
8
9 Status MLUBatchMatMulV2::CreateMLUOp(std::vector<MLUTensor *> &inputs,
10                                     std::vector<MLUTensor *> &outputs,
11                                     void *param) {
12   TF_PARAMS_CHECK(inputs.size() > 1, "Missing input");
13   TF_PARAMS_CHECK(outputs.size() > 0, "Missing output");
14   MLUTensor *in0 = inputs.at(0);
15   MLUTensor *in1 = inputs.at(1);
16   MLUTensor *output = outputs.at(0);
17
18   MLULOG(3) << "CreateBatchMatMulV2Op, input1: "
19             << lib::MLUTensorUtil(in0).DebugString()
20             << ", input2: " << lib::MLUTensorUtil(in1).DebugString()
21             << ", output: " << lib::MLUTensorUtil(output).DebugString();
22
23   MLUBatchMatMulV2OpParam *op_param =
24     static_cast<MLUBatchMatMulV2OpParam *>(param);
25
26   float scale_0 = op_param->scale_0_;
27   int pos_0 = op_param->pos_0_;
28   float scale_1 = op_param->scale_1_;
29   int pos_1 = op_param->pos_1_;
30   bool trans_flag = op_param->trans_flag_;
31   int dim_0 = op_param->dim_0_;
32   int dim_1 = op_param->dim_1_;
33   int m = op_param->m_;
34   int n = op_param->n_;
35   int k = op_param->k_;
36   cnmlCoreVersion_t core_version = CNML_MLU270;
37
38   bool* flags = (bool*)malloc(1 * sizeof(bool));
39   flags[0] = trans_flag;
40   extra_ = static_cast<void*>(flags);
41   //在这里对右矩阵完成转置操作
42   if (trans_flag) {
43     lib::MLUTensorUtil input_util(in1);
44     tensorflow::TensorShape input_shape_t = {dim_0, dim_1, n, k};
45     std::vector<int> input_shape_t_vec(input_shape_t.dims());
46     for (int i = 0; i < input_shape_t.dims(); ++i)
47       input_shape_t_vec[i] = input_shape_t.dim_size(i);
48     MLUTensor* input_transpose = nullptr;
49     TF_STATUS_CHECK(lib::CreateMLUTensor(&input_transpose, MLU_TENSOR,
50                                         input_util.dtype(), input_shape_t_vec));
51     MLUBaseOp* input_transpose_op = nullptr;
52     std::vector<int> input_perms = {0, 1, 3, 2};
53     TF_STATUS_CHECK(lib::CreateTransposeProOp(&input_transpose_op, in1,
54                                             input_transpose, input_perms.data(),
55                                             input_perms.size()));
56     base_ops_.push_back(input_transpose_op);
57     intmd_tensors_.push_back(input_transpose);
58     in1 = input_transpose;
59     input_util.Update(in1);
60   } else {
61     base_ops_.push_back(nullptr);
62     intmd_tensors_.push_back(nullptr);
63   }
64
65   cnmlPluginBatchMatMulV2OpParam_t bm_param;
66   TF_CNML_CHECK(cnmlCreatePluginBatchMatMulV2OpParam(&bm_param, scale_0, pos_0,
67                                                       scale_1, pos_1, dim_0, dim_1, m, n, k, core_version));

```

```
1 // tensorflow/stream_executor/mlu/mlu_stream.h
2 Status BatchMatMulV2(OpKernelContext* ctx,
3   Tensor* input1, Tensor* input2,
4   float scale_0, int pos_0, float scale_1,
5   int pos_1, int dim_0, int dim_1, int m,
6   int n, int k, Tensor* output) {
7   ops::MLUBatchMatMulV2OpParam op_param(scale_0, pos_0, scale_1,
8     pos_1, dim_0, dim_1, m, n, k);
9   return CommonOImpl<ops::MLUBatchMatMulV2>(ctx, {input1, input2},
10    {output}, static_cast<void*>(&op_param));
11 }
```

图 7.71 MLUStream 接口

```
1 // tensorflow/core/kernels/batch_matmul_v2_op_mlu.h
2 #ifndef TENSORFLOW_CORE_KERNELS_BATCH_MATMUL_V2_OP_MLU_H_
3 #define TENSORFLOW_CORE_KERNELS_BATCH_MATMUL_V2_OP_MLU_H_
4 #if CAMBRICON_MLU
5 ...
6 #include "tensorflow/stream_executor/mlu/mlu_stream.h"
7
8 namespace tensorflow {
9
10 template <typename T>
11 class MLUBatchMatMulV2 : public MLUOpKernel {
12 public:
13   explicit MLUBatchMatMulV2(OpKernelConstruction* context)
14     : MLUOpKernel(context) {
15     OP_REQUIRES_OK(context, context->GetAttr("adj_x", &adj_x_));
16     OP_REQUIRES_OK(context, context->GetAttr("adj_y", &adj_y_));
17     // position ans scale
18     ...
19     // input1_position
20     ...
21     // input1_scale
22     ...
23     // input2_position
24     ...
25     // input2_scale
26     ...
27   }
28
29   void ComputeOnMLU(OpKernelContext* ctx) override {
30     se::mlu::MLUStream* stream = static_cast<se::mlu::MLUStream*>(
31       ctx->op_device_context()->stream()->implementation());
32     ...
33     // 调用 stream 中的 BatchMatMulV2 函数
34     OP_REQUIRES_OK(ctx, stream->BatchMatMulV2(ctx, ...));
35   };
36
37 private:
38   bool adj_x_;
39   bool adj_y_;
40   // 其他数据成员
41   ...
42 };
43 } // namespace tensorflow
44 #endif // CAMBRICON_MLU
45 #endif // TENSORFLOW_CORE_KERNELS_BATCH_MATMUL_V2_OP_MLU_H_
```

图 7.72 MLUOpKernel 层接口

```

1 python run_squad.py \
2   --vocab_file=${BERT_BASE_DIR}/vocab.txt \
3   --bert_config_file=${BERT_BASE_DIR}/bert_config.json \
4   --predict_batch_size=8 \
5   --max_seq_length=128 \
6   --hidden_size=768 \
7   --output_dir=${OUTPUT_DIR} \
8   --export_frozen_graph=false \
9   --do_predict=true \
10  --predict_file=${SQUAD_DIR}/dev-v1.1.json
11 python ${SQUAD_DIR}/evaluate-v1.1.py \
12   ${SQUAD_DIR}/dev-v1.1.json \
13   ${OUTPUT_DIR}/predictions.json

```

图 7.73 BERT 推理

```

1 [config]
2 quantization_type = int16
3 device_mode = clean
4 int_op_list = FC, BatchMatMul
5
6 [model]
7 input_nodes = input_ids, input_mask, segment_ids
8 layer_num = 12
9 seq_length = 128
10 original_models_path = /path/to/frozen_model.pb
11 output_nodes = unstack
12 quantization_output_nodes = bert_plugin_op
13 scope_names = attention/self/query/MatMul, attention/self/key/MatMul,
14               attention/self/value/MatMul, attention/self/MatMul,
15               attention/self/MatMul_1, attention/output/dense/MatMul,
16               intermediate/dense/MatMul, output/dense/MatMul
17 post_process_name = MatMul
18 save_model_path = /path/to/output/frozen_model_int16.pb
19
20 [data]
21 do_lower_case = False
22 doc_stride = 128
23 max_query_length = 64
24 batch_size = 8
25 iters = 1
26 vocab_file = bert/uncased_L-12_H-768_A-12/vocab.txt
27 data_list = bert/squad/dev-v1.1.json

```

图 7.74 BERT 大算子配置文件

附录 A DLP 软件环境介绍

A.1 整体环境

DLP 硬件的整体软件环境如图 A.1 所示。整体包括 6 个部分：编程框架、高性能库、编程语言及编译器、运行时库及驱动、开发工具包及领域专用开发包等。上层的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）间接通过调用高性能库及运行时库来运行。离线方式通过直接调用运行时库，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。以下重点介绍高性能库、运行时库以及开发工具包等。

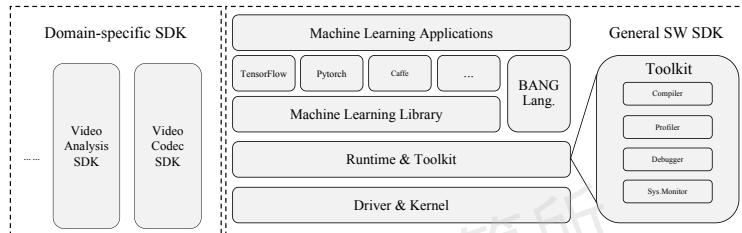


图 A.1 DLP 硬件的软件环境

A.2 运行时库 (CNRT)

DLP 的运行时库 CNRT 提供了面向 DLP 设备的用户层接口，用于完成包括设备管理、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。CNRT 包括的主要功能如表 A.1 所示。

表 A.1 DLP 运行时库功能介绍

功能模块	具体描述
设备管理	设备初始化、查询、指定等
内存管理	内存分配、释放、拷贝等
队列管理	队列 (Queue) 创建、销毁、同步等
通知管理	通知 (Notifier) 创建、销毁、同步等
任务管理	支持任务异步、并发、调度等

下述示例程序介绍了 mlp 算子的离线模型加载及计算过程

```

1 #include "cnrt.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 int offline_test(const char *name) {
7     // prepare model name

```

```
8  char fname[100] = "...";
9  strcat(fname, name);
10 strcat(fname, ".dlp");
11 // load model
12 cnrtModel_t model;
13 cnrtLoadModel(&model, fname);
14
15 cnrtInit(0);
16 unsigned dev_num;
17 cnrtGetDeviceCount(&dev_num);
18 if(dev_num == 0){
19     exit(-1);
20 }
21 cnrtDev_t dev;
22 cnrtGetDeviceHandle(&dev, 0);
23 cnrtSetCurrentDevice(dev);
24
25 // get model total memory
26 int64_t totalMem;
27 cnrtGetModelMemUsed(model, &totalMem);
28 printf("total memory used: %ld Bytes\n", totalMem);
29 // get model parallelism
30 int model_parallelism;
31 cnrtQueryModelParallelism(model, &model_parallelism);
32 printf("model parallelism: %d.\n", model_parallelism);
33 // load extract function
34 cnrtFunction_t function;
35 cnrtCreateFunction(&function);
36 cnrtExtractFunction(&function, model, name);
37 int inputNum, outputNum;
38 int64_t *inputSizeS, *outputSizeS;
39 cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
40 cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);
41 // prepare data on cpu
42 void **inputCpuPtrS = (void **)malloc(inputNum * sizeof(void *));
43 void **outputCpuPtrS = (void **)malloc(outputNum * sizeof(void *));
44 // allocate I/O data memory on MLU
45 void **inputMluPtrS = (void **)malloc(inputNum * sizeof(void *));
46 void **outputMluPtrS = (void **)malloc(outputNum * sizeof(void *));
47 // prepare input buffer
48 for (int i = 0; i < inputNum; i++) {
49     // converts data format when using new interface model
50     inputCpuPtrS[i] = malloc(inputSizeS[i]);
51     // malloc mlu memory
52     cnrtMalloc(&(inputMluPtrS[i]), inputSizeS[i]);
53     cnrtMemcpy(inputMluPtrS[i], inputCpuPtrS[i], inputSizeS[i], CNRT_MEM_TRANS_DIR_HOST2DEV);
54 }
55 // prepare output buffer
56 for (int i = 0; i < outputNum; i++) {
57     outputCpuPtrS[i] = malloc(outputSizeS[i]);
58     // malloc mlu memory
59     cnrtMalloc(&(outputMluPtrS[i]), outputSizeS[i]);
60 }
61 // prepare parameters for cnrtInvokeRuntimeContext
62 void **param = (void **)malloc(sizeof(void *) * (inputNum + outputNum));
63 for (int i = 0; i < inputNum; ++i) {
64     param[i] = inputMluPtrS[i];
65 }
66 for (int i = 0; i < outputNum; ++i) {
67     param[inputNum + i] = outputMluPtrS[i];
68 }
69 // setup runtime ctx
```

```

70 cnrtRuntimeContext_t ctx;
71 cnrtCreateRuntimeContext(&ctx, function, NULL);
72 // bind device
73 cnrtSetRuntimeContextDeviceId(ctx, 0);
74 cnrtInitRuntimeContext(ctx, NULL);
75 // compute offline
76 cnrtQueue_t queue;
77 cnrtRuntimeContextCreateQueue(ctx, &queue);
78 // invoke
79 cnrtInvokeRuntimeContext(ctx, param, queue, NULL);
80 // sync
81 cnrtSyncQueue(queue);
82 // copy mlu result to cpu
83 for (int i = 0; i < outputNum; i++) {
84     cnrtMemcpy(outputCpuPtrS[i], outputMluPtrS[i], outputSizeS[i], CNRT_MEM_TRANS_DIR_DEV2HOST
85     );
86 }
87 // free memory space
88 for (int i = 0; i < inputNum; i++) {
89     free(inputCpuPtrS[i]);
90     cnrtFree(inputMluPtrS[i]);
91 }
92 for (int i = 0; i < outputNum; i++) {
93     free(outputCpuPtrS[i]);
94     cnrtFree(outputMluPtrS[i]);
95 }
96 free(inputCpuPtrS);
97 free(outputCpuPtrS);
98 free(param);
99 cnrtDestroyQueue(queue);
100 cnrtDestroyRuntimeContext(ctx);
101 cnrtDestroyFunction(function);
102 cnrtUnloadModel(model);
103 cnrtDestroy();
104 return 0;
105 }
106 int main() {
107     printf("mlp offline test\n");
108     offline_test("mlp");
109 }
```

完成 CNRT 离线代码编写后，按照下面的示例进行编译和执行。

```

1 export NEUWARE=/path/to/neuware
2 g++ -c cnrt_mlp.cpp -I/path/to/neuware/include
3 g++ cnrt_mlp.o -o cnrt_mlp -L/path/to/neuware/lib64 -lcnrt
```

最后得到的输出结果如下：

```

1 ./cnrt_mlp
2
3 mlp offline test
4 CNRT: 4.2.1 fa5e44c
5 total memory used: 29851424 Bytes
6 model parallelism: 1.
```

A.3 高性能库 (CNML)

DLP 硬件的高性能库 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 中大量已优化的算子接口来实现其应用，同时可以根据自己的需求扩展已有算子。CNML 具有以下主要特性：

- 支持丰富的基本算子。已经支持的主要基本算子包括：常见神经网络运算（如卷积、池化和批规范化等），矩阵、向量及标量算子，循环神经网络算子（LSTM 和 RNN 等）等。
- 支持基本算子的融合。融合后的算子在编译期采用内存复用、片上存储管理和多核架构自适应等系列优化手段，显著提升融合算子的整体性能。
- 支持离线模型的生成。可以将编译好的算子（基本/融合）序列化到模型文件（离线模型）。该离线模型可以脱离编程框架和高性能库，采用底层运行时库接口直接运行。由于脱离了上次软件栈，离线模型的执行具有更好的性能和通用性。

除了提供针对特定算子预先优化好的高效实现外，CNML 中进一步提供了方便用户通过智能编程语言对高性能库算子进行扩展的插件 API (PluginOp)。通过 PluginOp，用户可以将自己用智能编程语言编写的算子“插入”到 CNML 高性能库中，与高性能库中原有算子的执行逻辑协同起来，支持包括算子融合和离线模型生成等多种功能。

下面将阐述如何使用 CNML 和 CNRT 库创建一个简单的 add 算子的过程。示例代码如下所示：

```
1 // define tensors: input, output
2 /*
3 * op name: add
4 * input_1 size: n x c x h x w
5 * input_2 size: n x c x h x w
6 * output size: n x c x h x w
7 */
8 #include "cnml.h"
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 int add_test() {
13     const cnmlCoreVersion_t coreVersion = CNML_MLU270;
14     cnrtInit(0);
15     unsigned dev_num;
16     cnrtGetDeviceCount(&dev_num);
17     if(dev_num == 0){
18         exit(-1);
19     }
20     cnrtDev_t dev;
21     cnrtGetDeviceHandle(&dev, 0);
22     cnrtSetCurrentDevice(dev);
23
24     // prepare data for pool
25     const int dimNum = 4;
26     const int n = 1, c = 32, h = 4, w = 4;
27     const int coreNum = 4;
28     // count input, filter, bias, output nums
29     int input_count_1 = n * h * w * c;
30     int input_count_2 = n * h * w * c;
31     int output_count = n * h * w * c;
32     float *input_cpu_ptr_1 = (float *)malloc(input_count_1 * sizeof(float));
33     float *input_cpu_ptr_2 = (float *)malloc(input_count_2 * sizeof(float));
34     float *output_cpu_ptr = (float *)malloc(output_count * sizeof(float));
```

```

35     unsigned int seed = 123;
36     for (int i = 0; i < input_count_1; i++) {
37         input_cpu_ptr_1[i] = ((rand_r(&seed) % 100 / 100.0) - 0.5) / 2;
38     }
39     for (int i = 0; i < input_count_2; i++) {
40         input_cpu_ptr_2[i] = (rand_r(&seed) % 100 / 100.0) - 0.5;
41     }
42     // set tensor shapes
43     int input_shape_1[] = {n, c, h, w};
44     int input_shape_2[] = {n, c, h, w};
45     int output_shape[] = {n, c, h, w};
46     // prepare input tensor 1
47     cnmlTensor_t input_tensor_1 = NULL;
48     cnmlCreateTensor_V2(&input_tensor_1, CNML_TENSOR);
49     cnmlSetTensorShape_V2(input_tensor_1, dimNum, input_shape_1, NULL);
50     cnmlSetTensorDataType(input_tensor_1, CNML_DATA_FLOAT32);
51     // prepare input tensor 2
52     cnmlTensor_t input_tensor_2 = NULL;
53     cnmlCreateTensor_V2(&input_tensor_2, CNML_TENSOR);
54     cnmlSetTensorShape_V2(input_tensor_2, dimNum, input_shape_2, NULL);
55     cnmlSetTensorDataType(input_tensor_2, CNML_DATA_FLOAT32);
56     // prepare output tensor
57     cnmlTensor_t output_tensor = NULL;
58     cnmlCreateTensor_V2(&output_tensor, CNML_TENSOR);
59     cnmlSetTensorShape_V2(output_tensor, dimNum, output_shape, NULL);
60     cnmlSetTensorDataType(output_tensor, CNML_DATA_FLOAT32);
61     // create add operator
62     cnmlBaseOp_t add_op;
63     cnmlCreateAddOp(&add_op, input_tensor_1, input_tensor_2, output_tensor);
64     // compile op
65     cnmlSetBaseOpCoreVersion(add_op, coreVersion);
66     cnmlSetBaseOpCorenum(add_op, coreNum);
67     cnmlCompileBaseOp_V2(add_op);
68     // mlu buffer ptr
69     void *input_mlu_ptr_1 = NULL;
70     void *input_mlu_ptr_2 = NULL;
71     void *output_mlu_ptr = NULL;
72     // malloc cnml tensor
73     cnrtMalloc(&input_mlu_ptr_1, input_count_1 * sizeof(float));
74     cnrtMalloc(&input_mlu_ptr_2, input_count_2 * sizeof(float));
75     cnrtMalloc(&output_mlu_ptr, output_count * sizeof(float));
76     // copy input to cnml buffer
77     cnrtMemcpy(input_mlu_ptr_1, input_cpu_ptr_1, input_count_1 * sizeof(float),
78                CNRT_MEM_TRANS_DIR_HOST2DEV);
79     cnrtMemcpy(input_mlu_ptr_2, input_cpu_ptr_2, input_count_2 * sizeof(float),
80                CNRT_MEM_TRANS_DIR_HOST2DEV);
81     // set cnrt queue
82     cnrtQueue_t queue;
83     cnrtCreateQueue(&queue);
84     cnmlComputeAddOpForward_V4(add_op, NULL, input_mlu_ptr_1, NULL, input_mlu_ptr_2, NULL,
85                                output_mlu_ptr, queue, NULL);
86     // wait for computing task over cnrtSyncQueue(queue);
87     // end of queue life cycle cnrtDestroyQueue(queue);
88     // copy output to cpu
89     cnrtMemcpy(output_cpu_ptr, output_mlu_ptr, output_count * sizeof(float),
90                CNRT_MEM_TRANS_DIR_DEV2HOST);
91     // dump mlu result to file mlu_output
92     cnmlDumpTensor2File_V2("mlu_output", output_tensor, output_cpu_ptr, false);
93     // delete op ptr
94     cnmlDestroyBaseOp(&add_op);
95     // delete cnml buffer
96     cnrtFree(input_mlu_ptr_1);

```

```

93     cnrtFree(input_mlu_ptr_2);
94     cnrtFree(output_mlu_ptr);
95     // delete cnml tensors
96     cnmlDestroyTensor(&input_tensor_1);
97     cnmlDestroyTensor(&input_tensor_2);
98     cnmlDestroyTensor(&output_tensor);
99     // delete pointers (including data pointers)
100    free(input_cpu_ptr_1);
101    free(input_cpu_ptr_2);
102    free(output_cpu_ptr);
103    return 0;
104}
105int main() {
106    printf("cnml add test\n");
107    add_test();
108    return 0;
109}

```

其主要分为 8 个步骤。

- 创建 CPU 端的张量，并准备 CPU 端的数据。
- 创建 MLU 端的张量，利用 MLU 端的张量做为输入创建 AddOp。
- 编译。
- 将 CPU 端的数据拷入到 MLU 端。
- 执行 MLU 端的计算过程。
- 将 MLU 端的结果拷口到 CPU 端。
- 计算结果存放在文件中。
- 释放 MLU 和 CPU 端的资源。

完成 CNML 示例代码编写后，按照以下方式进行编译和执行。

```

1 #编译
2 export NEUWARE=/path/to/neuware
3 g++ -c cnml_demo.cpp -I/path/to/neuware/include
4 g++ cnml_demo.o -o cnml_demo -L /path/to/neuware/lib64 -lcnrt -lcnml
5 #执行
6 ./cnml_demo
7 #输出
8 cnml add test
9 CNRT: 4.2.1 fa5e44c

```

A.4 开发工具包

除了上述基本软件模块外，DLP 还提供了多种工具方便用户进行状态监测及性能调优，典型的如应用级性能剖析工具、系统级性能监控工具和调试器等。以下重点介绍本节实验中用到的应用级性能剖析工具和系统级性能监控工具。

应用级性能剖析工具 (CNPerf) 以性能事件为基础，可以精确地获得用户程序中每个函数的执行细节信息。如：函数调用栈信息；用户程序或部分依赖库中函数的执行时间；Host

侧和 DLP 侧的内存占用开销；高性能库中算子的计算效率及 DDR 访存带宽；用户自定义 Kernel 函数的实际执行时间等。

CNPerf 命令行主要支持以下几个命令：

- **record**: 记录性能数据并保存到数据文件中。数据文件默认保存在 `dltrace_data` 文件夹中，该命令可以使用相关参数改变该数据文件所在的默认路径。
- **report**: 在终端上显示目标程序的总体信息。
- **replay**: 在终端上显示所有日志信息。
- **kernel**: 在终端上展示更多 pmu 数据。
- **show**: 不记录日志，直接将数据打印到终端。
- **monitor**: 提供旁路指令，查看 Jpu/Vpu 利用率，以及 Vpu 读写的带宽和累计值。
- **info**: 显示本次 record 运行环境相关的信息。
- 显示所有命令的帮助信息。用户执行该命令可以查看 CNPerf 支持的命令及其使用方法。

使用者可直接在 /usr/local/neuware/bin 下执行 `cnperf`，或通过将上述目录添加至 PATH 环境变量中，即可在任意位置执行。

下面以命令行使用方法为例，简要介绍如何使用 `cnperf` 进行性能分析。生成并查看性能数据的步骤如下所示：

- 使用 `record` 命令生成性能数据，并保存到数据文件中。数据文件默认保存到生成的 `dltrace_data` 文件夹下。
 - 以 `dltrace_data` 文件夹下的数据文件为输入，执行 `report` 命令查看性能数据信息。
 - 以 `dltrace_data` 文件夹下的数据文件为输入，执行 `replay` 命令显示所有监测日志信息。
 - 以 `dltrace_data` 文件夹下的数据文件为输入，执行 `kernel` 命令显示所有监测 pmu 性能信息。
- 使用 `show` 命令生成性能数据，不记录日志，直接在终端显示所有监测的性能信息。
- 使用 `monitor` 命令主要是用来提供旁路指令，查看 Jpu/Vpu 利用率，以及 Vpu 读写的带宽和累计值。

注意，CNPerf 跟踪的目标程序需要加`-pg` 参数编译，不加`-pg` 编译的目标程序 CNPerf 无法跟踪。举例如下：`gcc test.c -pg -o test`

下面对 `test` 可执行文件进行性能分析。

a. 通过指令跟踪 `test` 可执行文件：`cnperf-cli record test`。执行完毕后该目录会有 `dltrace_data` 文件夹生成。

b. 进入放置 `dltrace_data` 的目录，使用 `report` 命令查看 `dltrace_data` 数据信息：`cnperf-cli report`。得到如图 A.2 所示信息，包含被跟踪程序的线程号、Function 执行时间、调用函数次数、`kernel` 的计算效率、`kernel` 执行时计算单元对设备内存的读写数量与总带宽等信息。

c. 此外还可以分别执行：`cnperf-cli replay`；`cnperf-cli kernel`；`cnperf-cli monitor`；`cnperf-cli info` 等指令获取相应的信息。

系统级性能监控工具（CNMon）主要通过利用驱动读取寄存器的方式来搜集硬件的静态和动态执行信息。在 DLP 硬件上可以采集的信息包括：硬件设备型号、驱动版本号、设

PID	Total time	Self time	Calls	Function	
25150	918.722 us	918.722 us	228	cnrtInvokeFunction	
	698.348 us	698.348 us	269	cnmlBindConstData	
	666.434 us	666.434 us	228	cnrtCreateKernelHeapAllocInfo	
	623.647 us	623.647 us	228	cnrtUpdateBarrierInstV3	
	...				
	3.204 us	3.204 us	2	cnrtInvokeKernelRecordList_clear	
	3.044 us	3.044 us	1	cnmlDestroyConvFirstOpParam	
<hr/>					
Kernels Info:					
Pid	Duration	ComputeSpeed	IOSpeed	IOCount	Function
25150	138.00 us	7721 GOPS/s	4.973 GiB/s	736894	cnmlComputeConvFirstOpForward_V3[1]
	146.00 us	5137 GOPS/s	10.41 GiB/s	1631363	cnmlComputeBatchNormOpForward_V3[1]
	146.00 us	5137 GOPS/s	10.41 GiB/s	1631363	cnmlComputeScaleOpForward_V3[1]
	...				
	127.00 us	8366 GOPS/s	15.52 GiB/s	2116409	cnmlComputeMlpOpForward_V3[1]
	11.00 us	9500 GOPS/s	1.478 GiB/s	17454	cnmlComputeSoftmaxOpForward_V3[1]
<hr/>					
Max MLU: 0 MB					
DEVICE_TO_HOST size: 0MB					
HOST_TO_DEVICE size: 42.7614MB speed: 0.00514178 GiB/s					

图 A.2 CNPerf Report

备利用率、物理内存总量、虚拟内存总量、进程内存使用量、板卡功耗及温度、PCIe 信息等。

在正确安装 neuware-driver 后，即可在终端使用 cnmon 指令，得到如图 A.3 的界面。分别包含：板卡号（Card）、板卡名称（Name）、口扇转速比（Fan）、功率和峰值功率（Pwr）、驱动版本（Driver）、利用率（Util）、虚拟内存使用情况（vMemory-Usage）、是否为虚拟机（VF）、固件版本（Firmware）、板卡温度（Temp）、cndev 是否初始化（Initied）、物理内存使用情况（Memory-Usage）、Ecc 报错数量统计（Ecc-Error）

```
+-----+  
| CNMON 1.11.0 |  
+-----+  
| Card VF Name     Firmware | Initiated      Driver | Util       Ecc-Error |  
| Fan   Temp     Pwr:Usage/Cap |           Memory-Usage | vMemory-Usage |  
+=====+=====+=====+=====+=====+=====+  
| 0     /    MLU270  v0.2.0 | On            v2.2.0 | 0%          0 |  
| 19%  44C        22 W/ 150 W | 2705 MiB/ 16384 MiB | 10240 MiB/1048576 MiB |  
+-----+  
  
+-----+  
| Processes:  
| Card VF PID  Command Line | MLU Memory Usage |  
+=====+=====+=====+=====+  
| No running processes found |  
+-----+
```

图 A.3 CNMon 监控工具