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[?] 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速膨胀。

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：

```
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：

```
[preprocess]
mean = 123.68, 116.78, 103.94          ; 均值，顺序依次为 mean_r、 mean_g、 mean_b
std = 1.0                              ; 方差
color_mode = rgb                       ; 网络的输入图片是 rgb、bgr、grey
crop = 224, 224                        ; 前处理最终将图片处理为 224 * 224 大小
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

[config]
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
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
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）	计算输入张量 input 和卷积核 filter 的卷积，返回卷积计算的结果张量。	input：输入张量，仅支持 half、bfloat16、float32、float64 类型。 filter：卷积核，数据类型需与 input 一致。 strides：卷积步长。 padding：边界扩充，其值为"SAME" 或"VALID"。"SAME"表示对输入先进行边界扩充再进行卷积运算；"VALID"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。 use_cudnn_on_gpu：布尔值，缺省为 True。 data_format：输入和输出数据的数据格式，其值为"NHWC" 或"NCHW"。缺省为"NHWC"，表示数据存储格式为：[batch，height，width，channels]。 dilations：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。 name：可选参数，表示操作的名称。 filters：同 filter。
tf.nn.bias_add(value, bias, data_format=None, name=None）	对输入张量 value 加上偏置 bias，并返回一个与 value 相同类型的张量。	value：输入张量。其数据类型包括 float、double、int64、int32、uint8、int16、int8、complex6 或 complex128。 bias：一阶张量，其形状 (shape) 与 value 的最后一阶一致，数据类型需与 value 一致 (量化类型除外)。由于该函数支持广播形式，因此 value 可以有任意形状。 data_format：输入张量的数据格式。 name：可选参数，表示操作的名称。
tf.nn.relu(features, name=None）	对输入张量 features 计算 ReLU，返回一个与 features 相同数据类型的张量。	features：输入张量，其数据类型包括 float32、float64、int32、uint8、int16、int8、int64、bfloat16、uint16、half、uint32、uint64 或 qint8。 name：可选参数，表示操作的名称。
tf.nn.softmax(logits, axis=None, name=None, dim=None ）	对输入张量 logits 执行 softmax 激活操作，返回一个与 logits 相同数据类型、形状的张量。	logits：输入张量，其数据类型包括 half、float32、float64。 axis：执行 softmax 操作的维度，缺省为-1，表示最后一个维度。 name：可选参数，表示操作的名称。
tf.nn.max_pool(value, ksize, strides, padding, data_format='NHWC', name=None, input=None）	对输入张量 value 执行最大池化操作，返回操作的结果。	value：输入张量，其数据格式由 data_format 定义。 ksize：对输入张量的每个维度执行最大池化操作的窗口尺寸。 strides：对输入张量的每个维度执行最大池化操作的滑动步长。 padding：边界扩充，其值为"SAME" 或"VALID"。 data_format：输入和输出数据的数据格式，支持"NHWC"、"NCHW" 及"NCHW_VECT_C" 格式。 name：可选参数，表示操作的名称。 input：同 value。
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）	计算输入张量 value 和卷积核 filter 的转置卷积，返回计算的结果张量。	value：转置卷积计算的输入张量，数据类型为 float，数据格式可以是"NHWC" 或"NCHW"。 filter：卷积核，数据类型需与 value 一致。 output_shape：转置卷积的输出形状。 strides：卷积步长。 padding：边界扩充，其值为"SAME" 或"VALID"。 data_format：输入和输出数据的数据格式，其值为"NHWC" 或"NCHW"。缺省为"NHWC"，表示数据存储格式为：[batch，height，width，channels]。 name：可选参数，表示返回的张量名称。 input：同 value。 filters：同 filter。 dilations：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
scipy.io.loadmat(file_name, mdict=None, appendmat=True, **kwargs)	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)。	file_name：.mat 文件的名称。 mdict：可选参数，插入了.mat 文件所列变量的字典。 appendmat：可选参数，布尔类型。为 True 表示将.mat 扩展名添加到 file_name 之后。 其余参数含义请参考[14]。
scipy.misc.imread(name, flatten=False, mode=None)	从文件 name 中读入一张图像，将其处理成 ndarray 类型的数据并返回。	name：待读取的文件名称。 flatten：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。 mode：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。
scipy.misc.imresize(arr, size, interp='bilinear', mode=None)	对图像 arr 的尺寸进行缩放，返回处理后的 ndarray 类型数据。	arr：待缩放的图像，数据类型为 ndarray。 size：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。 interp：用于缩放的插值方法，其值可以是"nearest"、"lanczos"、"bilinear"、"bicubic" 或"cubic" 等。 mode：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。
scipy.misc.imsave(name, arr, format=None)	将 ndarray 类型的数组 arr 保存为图像 name。	name：输出的图像文件名称。 arr：待保存的 ndarray 类型数组。 format：保存的图像格式。

```
use_convfirst = False                ; 是否使用 convfirst
quantization_type = int8             ; 量化位宽，目前可选 int8 和 int16
debug = False
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
int_op_list = Conv, FC, LRN          ; 要量化的 layer 的类型，目前可量化 Conv、FC 和 LRN
channel_quantization = False         ; 是否使用分通道量化

[model]
output_tensor_names = Softmax:0      ; 输入 Tensor 的名字，可以是多个，以逗号隔开
original_models_path = ../vgg19.pb   ; 输入 pb
save_model_path = ../vgg19_int8.pb   ; 输出 pb
input_tensor_names = img_placeholder:0  ; 输出 Tensor 的名字，可以是多个，以逗号隔开

[data]
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
data_path = ./image_list             ; 数据文件存放路径
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：input=(input−mean)/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。

```
# file: evaluate_cpu.py
import scipy.misc
import numpy as np
import time
import tensorflow as tf

os.putenv('MLU_VISIBLE_DEVICES','')#设置MLU_VISIBLE_DEVICES="" 来屏蔽DLP

def load_image(path):
# 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
    mean = np.array([123.68, 116.779, 103.939])
    image = _____
    _____
    return image

def preprocess(image,mean):
    return image - mean

```

图 4.1　读取图像作为输入

```
# file: evaluate_cpu.py
def _conv_layer(input, weights, bias):
# 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
    _____

def _pool_layer(input):
# 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
    _____

```

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。

```
# file: evaluate_cpu.py
def net(data_path, input_image):
# 该函数定义VGG19网络结构，data_path为预训练好的模型文件，
    input_image为待分类的输入图像，该函数定义43层的VGG19网络结构
    net并返回该网络
    layers = (
        'conv1_1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1',

        'conv2_1', 'relu2_1', 'conv2_2', 'relu2_2', 'pool2',
        'conv3_1', 'relu3_1', 'conv3_2', 'relu3_2', 'conv3_3',
        'relu3_3', 'conv3_4', 'relu3_4', 'pool3',
        'conv4_1', 'relu4_1', 'conv4_2', 'relu4_2', 'conv4_3',
        'relu4_3', 'conv4_4', 'relu4_4', 'pool4',
        'conv5_1', 'relu5_1', 'conv5_2', 'relu5_2', 'conv5_3',
        'relu5_3', 'conv5_4', 'relu5_4', 'pool5',
        'fc6', 'relu6', 'fc7', 'relu7', 'fc8', 'softmax' )

    data = scipy.io.loadmat(data_path)
    weights = data['layers'][0]

    net = {}
    current = input_image
    for i, name in enumerate(layers):
        if name[:4] == 'conv':
        # TODO: 从模型中读取权重、偏置，执行卷积计算，结果存入 current
            ______________
        elif name[:4] == 'relu':
        # TODO: 执行 ReLU 计算，结果存入 current
            ______________
        # TODO: 完成其余层的定义，最终结果存入 current
        ______________

        net[name] = current

    assert len(net) == len(layers)
    return net

def preprocess(image,mean):
    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:

```
# file: evaluate_cpu.py
IMAGE_PATH = 'cat1.jpg' VGG_PATH = 'imagenet-vgg-verydeep-19.mat'

if __name__ == '__main__':
    input_image = load_image(IMAGE_PATH)

    with tf.Session() as sess:
        img_placeholder = tf.placeholder(tf.float32, shape
     =(1,224,224,3),
        name='img_placeholder')
        # TODO: 调用 net 函数，生成 VGG19 网络模型并保存在 nets 中
        nets = _____________
        for i in range(10):
            start = time.time()
            # TODO: 计算 nets
            _____________
            end = time.time()
            delta_time = end - start
            print("processing time: %s" % delta_time)

        prob = preds['softmax'][0]
        top1 = np.argmax(prob)
        print('Classification result: id = %d, prob = %f' % (top1,
     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 的机器学习编程库 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。

```
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：

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

```
# file: evaluate_dlp.py
import numpy as np
import struct
import os
import scipy.io
import time
import tensorflow as tf
from tensorflow.python.framework import graph_util #用于将模型文件保存为 pb 格式

if __name__ == '__main__':
    input_image = load_image(IMAGE_PATH)

    with tf.Session() as sess:
    # TODO: 代码见上一小节
        ---------------

        prob = preds['softmax'][0]
        top1 = np.argmax(prob)
        print('Classification result: id = %d, prob = %f' % (top1, prob[top1]))

        print("*** Start Saving Frozen Graph ***")
        # We retrieve the protobuf graph definition
        input_graph_def = sess.graph.as_graph_def()
        output_node_names = ["Softmax"]
        # We use a built-in TF helper to export variables to constant
        output_graph_def = graph_util.convert_variables_to_constants(
                sess,
                input_graph_def,
                output_node_names,
            )
        # Finally we serialize and dump the output graph to the filesystem
        with tf.gfile.GFile("vgg19.pb", "wb") as f:
                f.write(output_graph_def.SerializeToString())
        print("**** Save Frozen Graph Done ****")

```

图 4.5　将模型文件保存为 pb 格式

```
import struct
import os
import scipy.io
import time
import tensorflow as tf
from tensorflow.python.framework import graph_util

os.putenv('MLU_VISIBLE_DEVICES','0')#设置程序运行在DLP上

IMAGE_PATH = 'cat1.jpg'
VGG_PATH = 'vgg19_int8.pb'

if __name__ == '__main__':
    input_image = load_image(IMAGE_PATH)

    g = tf.Graph()

    # setting mlu configurations
    config = tf.ConfigProto(allow_soft_placement=True,
                            inter_op_parallelism_threads=1,
                            intra_op_parallelism_threads=1)
    config.mlu_options.data_parallelism = 1
    config.mlu_options.model_parallelism = 1
    config.mlu_options.core_num = 16 # 1 4 16
    config.mlu_options.core_version = "MLU270"
    config.mlu_options.precision = "int8"
    config.mlu_options.save_offline_model = False

    model = VGG_PATH

    with g.as_default():
        with tf.gfile.FastGFile(model,'rb') as f:
            graph_def = tf.GraphDef()
            graph_def.ParseFromString(f.read())
            tf.import_graph_def(graph_def, name='')

        with tf.Session(config=config) as sess:
            sess.run(tf.global_variables_initializer())
            input_tensor = sess.graph.get_tensor_by_name('img_placeholder:0')
            output_tensor = sess.graph.get_tensor_by_name('Softmax:0')

            for i in range(10):
                start = time.time()
                # TODO: 计算 output_tensor
                ______________
                end = time.time()
                delta_time = end - start
                print("Inference processing time: %s" % delta_time)

            prob = preds[0]
            top1 = np.argmax(prob)

            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。

```
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。

```
# 登录云平台
ssh root@xxx.xxx.xxx.xxx -p xxxxx
# 进入/opt/code_chap_4_student目录
cd /opt/code_chap_4_student
# 初始化环境
cd env
source env.sh
```

2. 代码实现

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

```
# 进入实验目录
cd exp_4_1_vgg19_student
# 补全 cpu 实现代码
vim stu_upload/evaluate_cpu.py
# 补全 mlu 实现代码
vim stu_upload/evaluate_mlu.py
```

3. CPU 运行

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

4. DLP 运行

```
# 对保存的 pb 模型进行量化
cd fppb_to_intpb
python fppb_to_intpb.py vgg19_int8.ini
# mlu 上运行
./run_mlu.sh
# 运行完整实验
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。

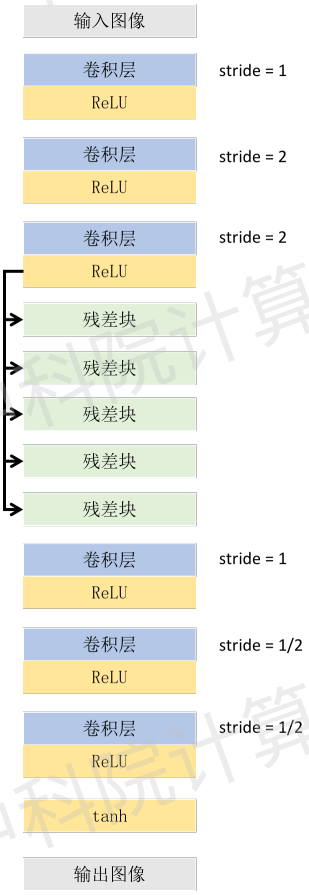

图 4.6　图像转换网络的网络结构

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 × 256 × 256
反射填充 (40 × 40)	3 × 336 × 336
32 × 9 × 9 卷积，步长 1	32 × 336 × 336
64 × 3 × 3 卷积，步长 2	64 × 168 × 168
128 × 3 × 3 卷积，步长 2	128 × 84 × 84
残差块，128 个卷积	128 × 80 × 80
残差块，128 个卷积	128 × 76 × 76
残差块，128 个卷积	128 × 72 × 72
残差块，128 个卷积	128 × 68 × 68
残差块，128 个卷积	128 × 64 × 64
64 × 3 × 3 卷积，步长 1/2	64 × 128 × 128
32 × 3 × 3 卷积，步长 1/2	32 × 256 × 256
3 × 9 × 9 卷积，步长 1	3 × 256 × 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 $\boldsymbol{x}$ 经过一个卷积层，再做 ReLU，然后经过另一个卷积层得到 $\boldsymbol{F}(\boldsymbol{x})$，再加上 $\boldsymbol{x}$ 得到输出 $\boldsymbol{H}(\boldsymbol{x}) = \boldsymbol{F}(\boldsymbol{x}) + \boldsymbol{x}$，然后做 ReLU 得到基本块的最终输出 $\boldsymbol{y}$。当输入 $\boldsymbol{x}$ 的维度与卷积输出 $\boldsymbol{F}(\boldsymbol{x})$ 的维度不同时，需要先对 $\boldsymbol{x}$ 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。

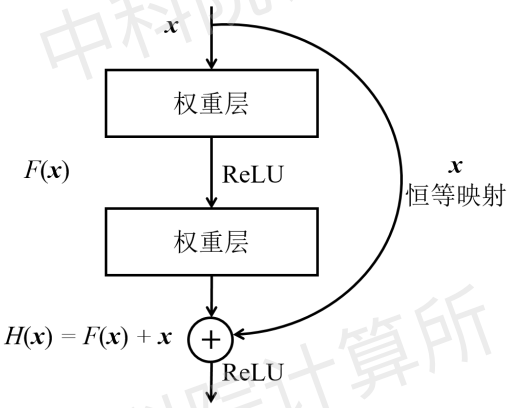

图 4.7　残差块结构

• **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。

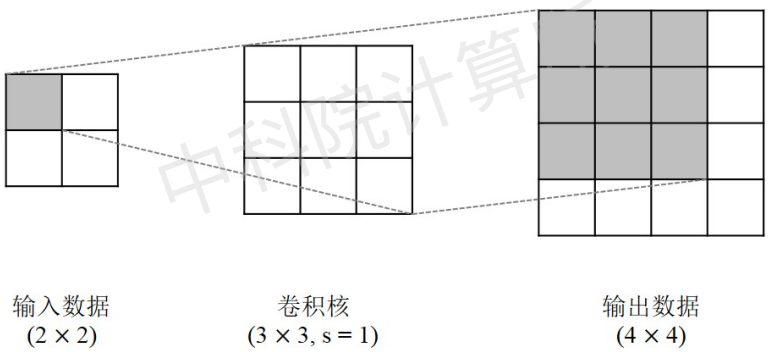

图 4.8　转置卷积

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 $\boldsymbol{x}$。

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

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

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

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

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

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

5. 上一步骤得到的 $\boldsymbol{y}$ 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)。

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

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。

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

- **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/。

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。

```
# file: src/utils.py
import scipy.misc
import numpy as np

def get_img(src, img_size = False):
    #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
    img = ________________
    ________________

    return img

```

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
```

```python
# file: evaluate_cpu.py
from __future__ import print_function
import sys
sys.path.insert(0, 'src')
import transform, NumPy as np, vgg, pdb, os
import scipy.misc
import tensorflow as tf
from utils import save_img, get_img, exists, list_files
from argparse import ArgumentParser
from collections import defaultdict
import time
import json
import subprocess
import numpy
BATCH_SIZE = 4
DEVICE = '/cpu:0'


os.putenv('MLU_VISIBLE_DEVICES','')#设置MLU_VISIBLE_DEVICES="" 来屏蔽DLP

def ffwd(data_in, paths_out, model, device_t='/gpu:0', batch_size=1):
    #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

    assert len(paths_out) > 0
    is_paths = type(data_in[0]) == str

    #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 1×256×256×3, 因此需将输入图像的形状调整为 256×256，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
    _____

    g = tf.Graph()
    config = tf.ConfigProto(allow_soft_placement=True,
    inter_op_parallelism_threads=1,
    intra_op_parallelism_threads=1)
    config.gpu_options.allow_growth = True
    with g.as_default():
        with tf.gfile.FastGFile(model,'rb') as f:
            graph_def = tf.GraphDef()
            graph_def.ParseFromString(f.read())
            tf.import_graph_def(graph_def, name='')

        with tf.Session(config=config) as sess:
            sess.run(tf.global_variables_initializer())
            input_tensor = sess.graph.get_tensor_by_name('X_content:0')
            output_tensor = sess.graph.get_tensor_by_name('add_37:0')
            batch_size = 1
            #TODO：读入的输入图像的数据格式为 HWC，还需要将其转换成 NHWC
            batch_shape = _____
            num_iters = int(len(paths_out)/batch_size)
            for i in range(num_iters):
            #分批次对输入图像进行处理
                pos = i * batch_size
                curr_batch_out = paths_out[pos:pos+batch_size]

                    #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
                    _____
                    start = time.time()
                    #TODO: 使用 sess.run 来计算 output_tensor
                    _preds = _____
                    end = time.time()
                    for j, path_out in enumerate(curr_batch_out):
                    #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
                        _____
```

```python
# file: evaluate_cpu.py
def ffwd_to_img(in_path, out_path, model, device='/cpu:0'):
    #该函数将上面的ffwd()函数用于图像的实时风格迁移
    paths_in, paths_out = [in_path], [out_path]
    ffwd(paths_in, paths_out, model, batch_size=1, device_t=device)

def main():
    #实时风格迁移预测函数主体
    #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
    parser = build_parser()
    opts = parser.parse_args()
    check_opts(opts)

    if not os.path.isdir(opts.in_path):
        #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
        if os.path.exists(opts.out_path) and os.path.isdir(opts.out_path):
            out_path = os.path.join(opts.out_path,os.path.basename(opts.in_path))
        else:
            out_path = opts.out_path

        #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
        ---------------
    else:
        #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
        #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
        files = list_files(opts.in_path)
        full_in = [os.path.join(opts.in_path,x) for x in files]
        full_out = [os.path.join(opts.out_path,x) for x in files]

        #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
        ---------------

if __name__ == '__main__':
    main()

```

图 4.11　实时风格迁移训练主函数

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

```

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。

```
# file: evaluate_mlu.py
import tensorflow as tf
...
#配置环境变量，设置程序运行在DLP上
os.putenv('MLU_VISIBLE_DEVICES','0')
...
#在生成session实例前，配置DLP参数
config = tf.ConfigProto(allow_soft_placement=True,
            inter_op_parallelism_threads=1,
                        intra_op_parallelism_threads=1)
config.mlu_options.data_parallelism = 1
config.mlu_options.model_parallelism = 1
config.mlu_options.core_num = 1
config.mlu_options.precision = "int8"
config.mlu_options.save_offline_model = True
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。

```
        # 登录云平台
        ssh root@xxx.xxx.xxx.xxx -p xxxxx
        # 进入/opt/code_chap_4_student目录
        cd /opt/code_chap_4_student
        # 初始化环境
        cd env
        source env.sh

```

2. 代码实现

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

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

```
# 补全 utils.py
vim src/utils.py
# 补全 cpu 实现代码
vim stu_upload/evaluate_cpu.py
# 补全 mlu 实现代码
vim stu_upload/evaluate_mlu.py

```

3. **CPU 运行**

```
# cpu 上运行
./run_cpu.sh

```

4. **DLP 运行**

```
# 对 pb 模型进行量化
cd fppb_to_intpb
python fppb_to_intpb.py udnie_int8.ini
# mlu 上运行
./run_mlu.sh
# 运行完整实验
python main_exp_4_2.py

```

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：

$$L = \mathrm{E}_{\boldsymbol{x}}\left[\lambda_1 L_{feat}(f_W(\boldsymbol{x}), \boldsymbol{y}_c) + \lambda_2 L_{style}(f_W(\boldsymbol{x}), \boldsymbol{y}_s)\right] \tag{4.3}$$

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：

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

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，$\phi(\boldsymbol{y})$ 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：

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

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

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

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

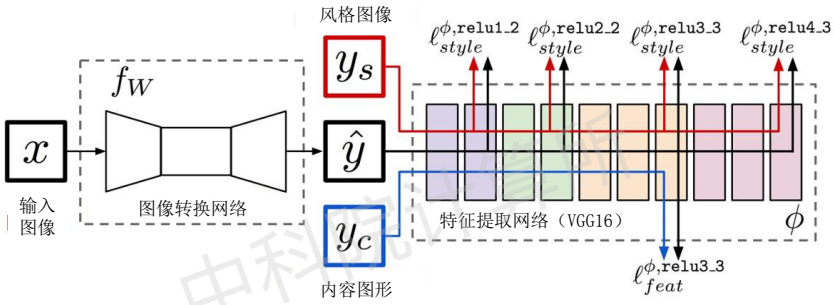

图 4.14　实时图像风格迁移算法的流程[15]

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**
最大池化					
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）$\boldsymbol{x}$ 输入到图像转换网络进行处理，输出生成图像 $\hat{\boldsymbol{y}}$；再将生成图像 $\hat{\boldsymbol{y}}$、风格图像 $\boldsymbol{y}_s$ 和内容图像 $\boldsymbol{y}_c = \boldsymbol{x}$ 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。

```python
# file: src/transform.py
import tensorflow as tf

def _conv_layer(net, num_filters, filter_size, strides, relu=True):
    #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

    #TODO：准备好权重的初值
    weights_init = _____

    #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
    _____

    #TODO：进行卷积计算
    net = _____

    #TODO：对卷积计算结果进行批归一化处理
    net = _____

    if relu:
    #TODO：对归一化结果进行 ReLU 操作
        net = _____

    return net

```

图 4.15　卷积层的定义

```python
# file:src/transform.py
def _residual_block(net, filter_size=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

    #TODO：调用上一步骤中实现的卷积层函数，实现残差块的计算
    _____

    return net

```

图 4.16　残差块的定义

```
# file:src/transform.py
def _conv_tranpose_layer(net, num_filters, filter_size, strides):
#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，
    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

    #TODO：准备好权重的初值
    weights_init = _____________
    _____________

    #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
    _____________

    #TODO：进行转置卷积计算
    net = _____________

    #TODO：对卷积计算结果进行批归一化处理
    net = _____________

    #TODO：对归一化结果进行 ReLU 操作
    net = _____________

    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。

```
# file：src/transform.py
def net(image):
#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

    #TODO：构建图像转换网络，每一层的输出作为下一层的输入
    conv1 = _____________
    conv2 = _____________
    _____________

    #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
    preds = _____________

    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。

```
# file: src/vgg.py
import tensorflow as tf
import numpy as np
import scipy.io
import pdb

def net(data_path, input_image):
    #定义特征提取网络，data_path为其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

    #TODO: 根据 VGG19 的网络结构定义每一层的名称
    layers = (
        'conv1_1', 'relu1_1', 'conv1_2', 'relu1_2', 'pool1',
        ________________
    )

    #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
    ________________

    net = {}
    current = input_image
    for i, name in enumerate(layers):
        kind = name[:4]
        if kind == 'conv'
            #TODO: 如果当前层为卷积层，则进行卷积计算，计算结果为 current
            ________________
        elif kind == 'relu':
            #TODO: 如果当前层为 ReLU 层，则进行 ReLU 计算，计算结果为 current
            ________________
        elif kind == 'pool':
            #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
            ________________
        net[name] = current

    assert len(net) == len(layers)
    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。

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

```
from __future__ import print_function
import functools
import vgg, pdb, time
import tensorflow as tf, NumPy as np, os
import transform
from utils import get_img

STYLE_LAYERS = ('relu1_1', 'relu2_1', 'relu3_1', 'relu4_1', 'relu5_1')
CONTENT_LAYER = 'relu4_2'
DEVICES = '/CPU:0'

def _tensor_size(tensor):
    #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
    from operator import mul
    return functools.reduce(mul, (d.value for d in tensor.get_shape()[1:]), 1)

def loss_function(net, content_features, style_features, content_weight, style_weight, tv_weight,
     preds, batch_size):
    #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

    batch_shape = (batch_size,256,256,3)

    #TODO: 计算内容损失
    content_size = _tensor_size(content_features[CONTENT_LAYER])*batch_size
    assert _tensor_size(content_features[CONTENT_LAYER]) == _tensor_size(net[CONTENT_LAYER])
    content_loss = _____________

    #计算风格损失
    style_losses = []
    for style_layer in STYLE_LAYERS:
        layer = net[style_layer]
        bs, height, width, filters = map(lambda i:i.value,layer.get_shape())
        size = height * width * filters
        feats = tf.reshape(layer, (bs, height * width, filters))
        feats_T = tf.transpose(feats, perm=[0,2,1])
        grams = tf.matmul(feats_T, feats) / size
        style_gram = style_features[style_layer]
        #TODO: 计算 style_losses
        _____________
    style_loss = style_weight * functools.reduce(tf.add, style_losses) / batch_size

    #使用全变分正则化方法定义损失函数tv_loss
    tv_y_size = _tensor_size(preds[:,1:,:,:])
    tv_x_size = _tensor_size(preds[:,:,1:,:])
    #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 L^2 范数 x_tv 和 y_tv
    _____________
    tv_loss = tv_weight*2*(x_tv/tv_x_size + y_tv/tv_y_size)/batch_size

    loss = content_loss + style_loss + tv_loss
    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。

```
# file: optimize.py
from __future__ import print_function
import functools
import vgg, pdb, time
import tensorflow as tf, NumPy as np, os
import transform
from utils import get_img

STYLE_LAYERS = ('relu1_1', 'relu2_1', 'relu3_1', 'relu4_1', 'relu5_1')
CONTENT_LAYER = 'relu4_2'
DEVICES = '/CPU:0'

def optimize(content_targets, style_target, content_weight, style_weight,
             tv_weight, vgg_path, epochs=2, print_iterations=1000,
             batch_size=4, save_path='saver/fns.ckpt', slow=False,
             learning_rate=1e-3, debug=True):
    #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、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VGG19网络参数的文件路径
    if slow:
        batch_size = 1
    mod = len(content_targets) % batch_size
    if mod > 0:
        print("Train set has been trimmed slightly..")
        content_targets = content_targets[:-mod]

    #风格特征预处理
    style_features = {}
    batch_shape = (batch_size,256,256,3)
    style_shape = (1,) + style_target.shape
    print(style_shape)

    with tf.Graph().as_default(), tf.device(DEVICES), tf.Session() as sess:
        #使用NumPy库在CPU上处理

        #TODO: 使用占位符来定义风格图像 style_image
        style_image = _____

        #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
        _____

        #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
        style_pre = np.array([style_target])
        for layer in STYLE_LAYERS:
            features = net[layer].eval(feed_dict={style_image:style_pre})
            features = np.reshape(features, (-1, features.shape[3]))
            gram = np.matmul(features.T, features) / features.size
            style_features[layer] = gram

        #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
        _____

        #提取内容特征对应的网络层
        content_features = {}
        content_net = vgg.net(vgg_path, X_pre)
        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                             _____________
```

```
                #将相关计算结果返回
                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]。

```
# file: style.py
from __future__ import print_function
import sys, os, pdb
sys.path.insert(0, 'src')
import numpy as np, scipy.misc
from optimize import optimize
from argparse import ArgumentParser
from utils import save_img, get_img, exists, list_files
import evaluate

os.putenv('MLU_VISIBLE_DEVICES','')#设置MLU_VISIBLE_DEVICES="" 来屏蔽DLP
CONTENT_WEIGHT = 7.5e0
STYLE_WEIGHT = 1e2
TV_WEIGHT = 2e2

LEARNING_RATE = 1e-3
NUM_EPOCHS = 2
CHECKPOINT_DIR = 'checkpoints'
CHECKPOINT_ITERATIONS = 2000
VGG_PATH = 'data/imagenet-vgg-verydeep-16.mat'
TRAIN_PATH = 'data/train2014'
BATCH_SIZE = 4
DEVICE = '/cpu:0'
FRAC_GPU = 1

def _get_files(img_dir):
    #读入内容图像目录下的所有图像并返回
    files = list_files(img_dir)
    return [os.path.join(img_dir,x) for x in files]

def main():
    #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
    parser = build_parser()
    options = parser.parse_args()
    check_opts(options)

    #TODO: 获取风格图像 style_target 以及内容图像数组 content_targets
    ---------------

    if not options.slow:
        content_targets = _get_files(options.train_path)
    elif options.test:
        content_targets = [options.test]

    kwargs = {
        "epochs":options.epochs,
        "print_iterations":options.checkpoint_iterations,
        "batch_size":options.batch_size,
        "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。

```
        "learning_rate":options.learning_rate
    }

    if options.slow:
        if options.epochs < 10:
            kwargs['epochs'] = 1000
        if options.learning_rate < 1:
            kwargs['learning_rate'] = 1e1

    args = [
        content_targets,
        style_target,
        options.content_weight,
        options.style_weight,
        options.vgg_path
    ]

    for preds, losses, i, epoch in optimize(*args, **kwargs):
        style_loss, content_loss, tv_loss, loss = losses

        print('Epoch %d, Iteration: %d, Loss: %s' % (epoch, i, loss))
        to_print = (style_loss, content_loss, tv_loss)
        print('style: %s, content:%s, tv: %s' % to_print)

    ckpt_dir = options.checkpoint_dir
    print("Training complete.\n")

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

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]。

```
python style.py --checkpoint-dir ckp_temp \
                --style examples/style/rain_princess.jpg \
                --train-path data/train2014_small \
                --content-weight 1.5e1 \
                --checkpoint-iterations 100 \
                --epochs 2 \
                --batch-size 4 \
                --type 0
```

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。

```
# 登录云平台
ssh root@xxx.xxx.xxx.xxx -p xxxxx
# 进入/opt/code_chap_4_student目录
cd /opt/code_chap_4_student
# 初始化环境
cd env
source env.sh

```

2. 代码实现

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

```
# 进入实验目录
cd exp_4_3_fast_style_transfer_train_student
# 补全 utils.py
vim src/utils.py
# 补全 transform.py
vim src/transform.py
# 补全 vgg.py
vim src/vgg.py
# 补全 optimize.py
vim src/optimize.py
# 补全训练主函数 style.py
vim style.py

```

3. 运行实验

```
# 运行完整实验
python main_exp_4_3.py
# 单独进行训练过程
./run_style.sh

```

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：

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

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 **X** 和 **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：

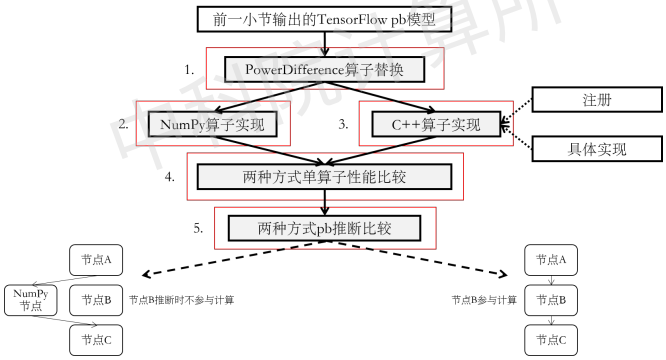

图 4.20　自定义算子实验流程图

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：

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

```
# file: pb_to_pbtxt.py
import argparse
import tensorflow as tf
from tensorflow.core.framework import graph_pb2

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('input_pb',help='input pb to be converted')
    parser.add_argument('output_pbtxt',help='output pbtxt generated')
    args = parser.parse_args()
    with tf.Session() as sess:
        with tf.gfile.FastGFile(args.input_pb, 'rb') as f:
            graph_def = graph_pb2.GraphDef()
            graph_def.ParseFromString(f.read())
            #tf.import_graph_def(graph_def)
        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。

```
node {
  name: "moments_15/PowerDifference_z"
  op: "Placeholder"
  attr {
    key: "dtype"
    value {
      type: DT_FLOAT
    }
  }
  attr {
    key: "shape"
    value {
      shape {
        unknown_rank: true
      }
    }
  }
}
```

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。

```
node {
  name: "moments_15/PowerDifference"
  op: "PowerDifference"
  input: "Conv2D_13"
  input: "moments_15/StopGradient"
  input: "moments_15/PowerDifference_z"
  attr {
    key: "T"
    value {
      type: DT_FLOAT
    }
  }
}
```

图 4.23　pbtxt 修改节点属性

```
node {
    name: "moments_15/variance"
    op: "Mean"
    input: "moments_15/PowerDifference"
    input: "moments_15/variance/reduction_indices"
    attr {
        key: "T"
        value {
        type: DT_FLOAT
        }
    }
    attr {
        key: "Tidx"
        value {
        type: DT_INT32
        }
    }
    attr {
        key: "keep_dims"
        value {
        b: true
        }
    }
}
```

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：

```
# file: power_diff_NumPy.py
import numpy as np
def power_diff_NumPy(input_x,input_y,input_z):
    #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
     size相同，input_y除了最后的维度，其余dim size均为1。
    x_shape = np.shape(input_x)
    y_shape = np.shape(input_y)
    x = np.reshape(input_x,(-1,y_shape[-1]))
    x_new_shape = np.shape(x)
    y = np.reshape(input_y,(-1))
    output = []
    #TODO：通过 for 循环完成计算，每次循环计算 y 个数的 PowerDifference 操作
```

```
#file: pbtxt_to_pb.py
import argparse
import tensorflow as tf
from tensorflow.core.framework import graph_pb2
from google.protobuf import text_format

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('input_pbtxt',help='input pbtxt to be converted')
    parser.add_argument('output_pb',help='output pb generated')
    args = parser.parse_args()
    with tf.Session() as sess:
        with tf.gfile.FastGFile(args.input_pbtxt, 'rb') as f:
        graph_def = graph_pb2.GraphDef()
        new_graph_def=text_format.Merge(f.read(), graph_def)
        tf.train.write_graph(new_graph_def, './', args.output_pb, as_text=False)
```

图 4.25　pbtxt-pb 转换工具

```
    for i in range(x_new_shape[0]):
        ___________________
    ___________________
    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：

```
# file: cwise_op_power_difference.cc
template <typename T>
class PowerDifferenceOp : public OpKernel {
  public:
    explicit PowerDifferenceOp(OpKernelConstruction* context)
      : OpKernel(context) {}

    void Compute(OpKernelContext* context) override {
      const Tensor& input_x_tensor = context->input(0);
      const Tensor& input_y_tensor = context->input(1);
      const Tensor& input_pow_tensor = context->input(2);

      const Eigen::ThreadPoolDevice& device = context->eigen_device<Eigen::ThreadPoolDevice>();
      //BCast部分，调用了TensorFlow自有的BCast算子，确保x和y的shape一致
      BCast bcast(BCast::FromShape(input_y_tensor.shape()), BCast::FromShape(input_x_tensor.shape
     ()),
                  /*fewer_dims_optimization=*/true);

      Tensor* output_tensor = nullptr;
      TensorShape output_shape = BCast::ToShape(bcast.output_shape());

```

```
    OP_REQUIRES_OK(context,
                   context->allocate_output(0, output_shape, &output_tensor));

    Tensor input_x_broad(input_x_tensor.dtype(), output_shape);
    Tensor input_y_broad(input_y_tensor.dtype(), output_shape);

    ......

    auto input_x = input_x_broad.flat<T>();
    auto input_y = input_y_broad.flat<T>();
    auto input_pow = input_pow_tensor.flat<T>();
    auto output = output_tensor->flat<T>();

    const int N = input_x.size();
    const int POW = input_pow(0);
    float tmp = 0;
    //实际计算部分
    for (int i = 0; i < N; i++) {
      //output(i) = (input_x(i) - input_y(i)) * (input_x(i) - input_y(i));
      tmp = input_x(i) - input_y(i);
      output(i) = tmp;
      for (int j = 0; j < POW - 1; j++){
        output(i) = output(i) * tmp;
      }
    }
  }
};
```

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。

```
REGISTER_OP("PowerDifference")
    .Input("x: T")
    .Input("y: T")
    .Input("pow: T")
    .Output("z: T")
    .Attr(
        "T: {bfloat16, float, half, double, int32, int64, complex64, "
        "complex128}")
    .SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
      c->set_output(0, c->input(0));
      c->set_output(0, c->input(1));
      c->set_output(0, c->input(2));
      return Status::OK();
    });
```

图 4.26　TensorFlow 注册 (1)

```
REGISTER_KERNEL_BUILDER(                                \
    Name("PowerDifference").Device(DEVICE_CPU),\
    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。

```
filegroup(
    name = "android_extended_ops_group1",
    srcs = [
        "argmax_op.cc",
        ......
        "cwise_op_power_difference.cc",
        //剩余注册文件
    ],
)
......
tf_kernel_library(
    name = "cwise_op",
    prefix = "cwise_op",
    deps = MATH_DEPS,
    hdrs = [
        "broadcast_to_op.h"
        ] + ......,
)
```

图 4.28　修改 BUILD 文件

```
#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。
bazel version
#2.激活环境
source env.sh
#3.执行编译脚本
cd /path/to/tensorflow/source
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：

```python
# file: power_difference_test_cpu.py
import numpy as np
import os
import time
import tensorflow as tf
from power_diff_NumPy import *
np.set_printoptions(suppress=True)

def power_difference_op(input_x,input_y,input_pow):
    with tf.Session() as sess:
        # TODO: 完成 TensorFlow 接口调用

        ______________________
        return sess.run(out, feed_dict = {...})

def main():
    start = time.time()
    input_x = np.loadtxt("./data/in_x.txt",delimiter=',')
    input_y = np.loadtxt("./data/in_y.txt")
    input_pow = np.loadtxt("./data/in_z.txt") #some specific number
    output = np.loadtxt("./data/out.txt")
    end = time.time()
    print("load data cost "+ str((end-start)*1000) + "ms" )
    # test C++ PowerDifference CPU Op
    start = time.time()
    res = power_difference_op(input_x, input_y, input_pow)
    end = time.time()
    print("comput C++ op cost "+ str((end-start)*1000) + "ms" )
    err = sum(abs(res - output))/sum(output)
    print("C++ op err rate= "+ str(err*100))
    # test NumPy PowerDifference Op
    start = time.time()
    res = power_diff_NumPy(input_x, input_y, input_pow)
    end = time.time()
    print("comput NumPy op cost "+ str((end-start)*1000) + "ms" )
    err = sum(abs(res - output))/sum(output)
    print("NumPy op err rate= "+ str(err*100))
if __name__ == '__main__':
    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：

```
node {
  name: "moments_15/PowerDifference"
  op: "Placeholder"
  attr {
    key: "dtype"
    value {
      type: DT_FLOAT
    }
  }
```

```
  attr {
    key: "shape"
    value {
      shape {
        unknown_rank: true
      }
    }
  }
}
```

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：

```
# file: transform_cpu.py
def run_NumPy_pb():
    args = parse_arg()
    config = tf.ConfigProto(allow_soft_placement=True,
                inter_op_parallelism_threads=1,
                            intra_op_parallelism_threads=1)
    model_name = os.path.basename(args.NumPy_pb).split(".")[0]
    image_name = os.path.basename(args.image).split(".")[0]

    g = tf.Graph()
    with g.as_default():
        with tf.gfile.FastGFile(args.NumPy_pb,'rb') as f:
            graph_def = tf.GraphDef()
            graph_def.ParseFromString(f.read())
            tf.import_graph_def(graph_def, name='')
        img = cv.imread(args.image)
        X = cv.resize(img, (256, 256))
        with tf.Session(config=config) as sess:
            sess.graph.as_default()
            sess.run(tf.global_variables_initializer())

            # TODO: 根据输入名称获得输入的 tensor
            input_tensor = ___________________
            # TODO: 获取两个输出节点，作为 NumPy 算子的输入
            out_tmp_tensor_1 = ___________________
            out_tmp_tensor_2 = ___________________
            # 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
            input_x,input_y =___________________
            input_pow = 2 #幂指数参数，可设为其它值，此处设置为2
            output = power_diff_NumPy(input_x, input_y, input_pow)

            # TODO: 根据添加的输入节点名称获得输入 tensor
            input_tensor_new = ___________________
            # TODO: 完整推断最终输出的 tensor
            output_tensor = ___________________
            # 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
            ret =___________________
            #结果保存
            img1 = tf.reshape(ret,[256,256,3])
            img_NumPy = img1.eval(session=sess)
            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：

```
# file: transform_cpu.py
def run_ori_power_diff_pb(ori_power_diff_pb, image):
    config = tf.ConfigProto(allow_soft_placement=True,
                inter_op_parallelism_threads=1,
                            intra_op_parallelism_threads=1)
    model_name = os.path.basename(ori_power_diff_pb).split(".")[0]
    image_name = os.path.basename(image).split(".")[0]

    g = tf.Graph()
    with g.as_default():
        with tf.gfile.FastGFile(ori_power_diff_pb,'rb') as f:
            graph_def = tf.GraphDef()
            graph_def.ParseFromString(f.read())
            tf.import_graph_def(graph_def, name='')
        img = cv.imread(image)
        X = cv.resize(img, (256, 256))
        with tf.Session(config=config) as sess:
            # TODO: 完成 PowerDifference pb 模型的推断
            ------------------

            start_time = time.time()
            ret =sess.run(...)
            end_time = time.time()
            print("C++ inference(CPU) time is: ",end_time-start_time)
            img1 = tf.reshape(ret,[256,256,3])
            img_NumPy = img1.eval(session=sess)
            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。

```
        # 登录云平台
        ssh root@xxx.xxx.xxx.xxx -p xxxxx
        # 进入/opt/code_chap_4_student目录
        cd /opt/code_chap_4_student
        # 初始化环境
        cd env
        source env.sh

```

2. 代码实现

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

```
# 进入实验目录
cd exp_4_4_custom_tensorflow_op_student
# 修改模型
python pb_to_pbtxt.py models/pb_models/udnie.pb udnie.pbtxt
vim udnie.pbtxt
# c++ 算子模型
python pbtxt_to_pb.py udnie.pbtxt udnie_power_diff.pb
# numpy 算子模型
python pbtxt_to_pb.py udnie.pbtxt udnie_power_diff_numpy.pb
# 补全 power_diff_numpy.py
vim stu_upload/power_diff_numpy.py
# 补全 cwise_op_power_difference.cc
vim tf-implementation/cwise_op_power_difference.cc
# 算子集成
cp tf-implementation/cwise_op_power_difference.* /opt/code_chap_4_student/env/tensorflow-v1.10/tensorflow/core/kernels/
vim /opt/code_chap_4_student/env/tensorflow-v1.10/tensorflow/core/kernels/BUILD
# 编译 tensorflow
/opt/code_chap_4_student/env/tensorflow-v1.10/build_tensorflow-v1.10_mlu.sh
# 补全 power_difference_test_cpu.py
vim stu_upload/power_difference_test_cpu.py
# 补全 transform_cpu.py
vim stu_upload/transform_cpu.py

```

3. 运行实验

```
# 将 tensorflow_mlu-1.14.0-cp27-cp27mu-linux_x86_64.whl, udnie_power_diff.pb 和 udnie_power_diff_numpy.pb 复制到 stu_upload 目录下
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/
cp udnie_power_diff.pb stu_upload/
cp udnie_power_diff_numpy stu_upload/
# 运行完整实验
./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

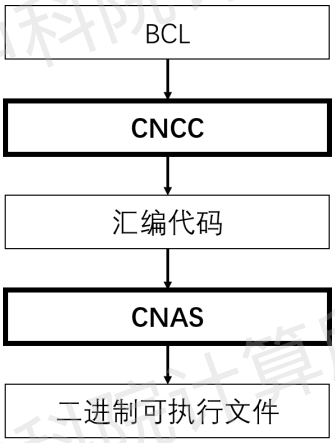

图 5.1　CNCC

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-ldram	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.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:
cncc recursion.mlu -o recursion.o --bang-mlu-arch=MLU270 -g -O0
#2.然后继续编译得到可运行二进制文件:
g++ recursion.o main.cpp -o quick_sort -lcnrt -I${DLP_INC} -L${DLP_LIB}
#3.在有DLP板卡的机器上使用CNGDB打开quick_sort程序:
cngdb quick_sort
#4.用break命令，在第x行添加断点:
(cn-gdb) b recursion.mlu :x
#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)
(cn-gdb) r
#6.用 print 命令，分别查看第一次调用x行函数时的三个实参:
(cn-gdb) p input1
(cn-gdb) p input2
(cn-gdb) p input3
#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:
(cn-gdb) c
#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:
(cn-gdb) s
#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:
(cn-gdb) info args
#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:
(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。

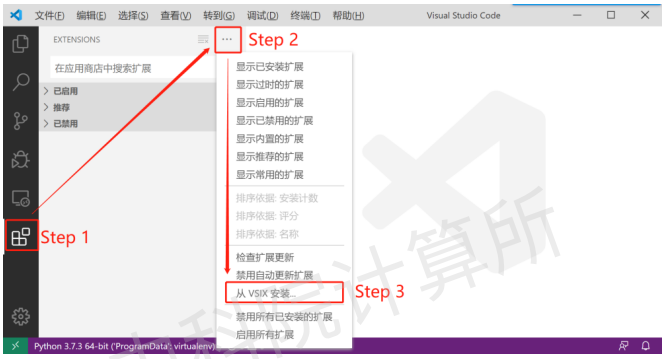

图 5.4　CNStudio 安装流程图

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

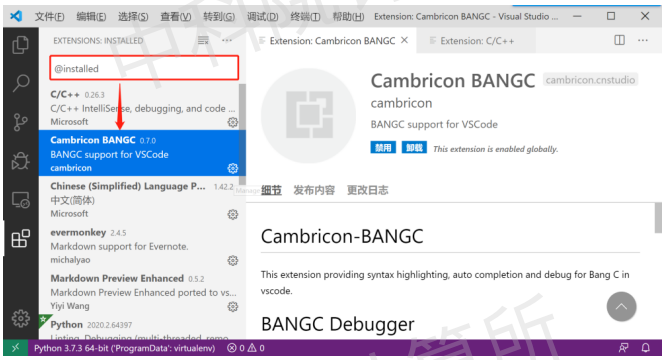

图 5.5　CNStudio 安装完成

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。

```
CNML_DLL_API cnmlStatus_t cnmlCreatePluginOp(cnmlBaseOp_t *op,
                                             const char *name,
                                             void *kernel,
                                             cnrtKernelParamsBuffer_t params,
                                             cnmlTensor_t *input_tensors,
                                             int input_num,
                                             cnmlTensor_t *output_tensors,
                                             int output_num,
                                             cnmlTensor_t *statics_tensor,
                                             int static_num);

CNML_DLL_API cnmlStatus_t cnmlComputePluginOpForward_V4(cnmlBaseOp_t op,
                                                        cnmlTensor_t input_tensors[],
                                                        void *inputs[],
                                                        int input_num,
                                                        cnmlTensor_t output_tensors[],
                                                        void *outputs[],
                                                        int output_num,
                                                        cnrtQueue_t queue,
                                                        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。Decive 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

```
Cambricon-CNPlugin-MLU270
├── build
├── build_aarch64.sh
├── build_cnplugin.sh
├── CMakeLists.txt
├── common
│   ├── include
│   │   ├── cnplugin.h
├── pluginops
│   ├── PluginYolov3DetectionOutputOp
│   │   ├── plugin_yolov3_detection_output_kernel_v2.h
│   │   ├── plugin_yolov3_detection_output_kernel_v2.mlu
│   │   └── plugin_yolov3_detection_output_op.cc
├── README.md
└── 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。

```

struct cnmlPluginAddpadOpParam {
  int batch_size;
  int src_h;
  int src_w;
  int dst_h;
  int dst_w;
  int type_uint8;
  int type_yuv;
};

cnmlStatus_t cnmlCreatePluginAddpadOpParam(cnmlPluginAddpadOpParam_t *param_ptr,
                                           int batch_size,
                                           int src_h,
                                           int src_w,
                                           int dst_h,
                                           int dst_w,
                                           int type_uint8,
                                           int type_yuv) {
  *param_ptr = new cnmlPluginAddpadOpParam();
  (*param_ptr)->batch_size = batch_size;
  (*param_ptr)->src_h = src_h;
  (*param_ptr)->src_w = src_w;
  (*param_ptr)->dst_h = dst_h;
  (*param_ptr)->dst_w = dst_w;
  (*param_ptr)->type_uint8 = type_uint8;
  (*param_ptr)->type_yuv = type_yuv;
  return CNML_STATUS_SUCCESS;
}

cnmlStatus_t cnmlDestroyPluginAddpadOpParam(cnmlPluginAddpadOpParam_t param) {
  delete param;
  return CNML_STATUS_SUCCESS;
}
```

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

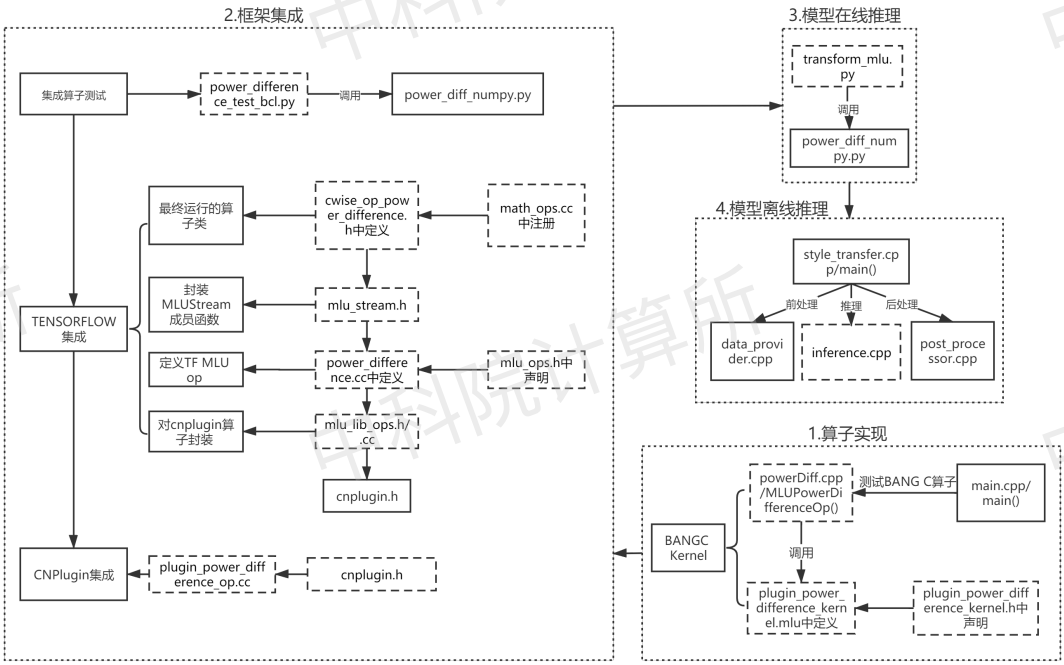

图 5.9　具体实验内容

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，

```
//filename: plugin_power_difference_kernel.mlu
//定义常量
#define ONELINE 256

__mlu_entry__ void PowerDifferenceKernel(half* input1, half* input2, int pow, half*
    output, int len)
{
  int quotient = len / ONELINE;
  __bang_printf("%d %d\n", pow, len);
  int rem = len % ONELINE;

  //声明NRAM空间
  __nram__ half input1_nram[ONELINE];
  __nram__ half input2_nram[ONELINE];

  if ( rem != 0)
  {
    quotient +=1;
  }
  for (int i = 0; i < quotient; i++)
  {
    //内存拷贝：从GDRAM的（input1 + i * ONELINE）位置开始，拷贝ONELINE * sizeof(half)大
     小的数据到input1_nram空间中。
    __memcpy(_____________________________);
    __memcpy(_____________________________);
    //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
     中。
    __bang_sub(___________________________);
    //NRAM中两个数据块的数据拷贝操作
    __memcpy(_____________________________);
    for (int j = 0; j < pow - 1; j++)
    {
      //按元素相乘操作
      __bang_mul(_________________________);
    }
    //内存拷贝：从NRAM中将计算的结果拷出至GDRAM中。
    __memcpy(_____________________________);
  }
}
```

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。

```cpp
//filename: powerDiff.cpp
#include <stdlib.h>
#include "cnrt.h"
#include "cnrt_data.h"
#include "stdio.h"
#ifdef __cplusplus
extern "C" {
#endif
void PowerDifferenceKernel(half* input1, half* input2, int32_t pow, half* output,
     int32_t len);
#ifdef __cplusplus
}
#endif
void PowerDifferenceKernel(half* input1, half* input2, int32_t pow, half* output,
     int32_t len);

int MLUPowerDifferenceOp(float* input1,float* input2, int pow, float*output, int dims_a)
     {
    //some definition
    //prepare data
    if (CNRT_RET_SUCCESS != cnrtMalloc((void**)&mlu_input1, dims_a * sizeof(half))) {
        printf("cnrtMalloc Failed!\n");
        exit(-1);
    }
    ...
    //kernel parameters
    cnrtKernelParamsBuffer_t params;
    cnrtGetKernelParamsBuffer(&params);
    cnrtKernelParamsBufferAddParam(params, &mlu_input1, sizeof(half*));
    ...
    //启动Kernel
    cnrtInvokeKernel_V2(_____________________________);
    //将计算结果拷回Host
    cnrtMemcpy(_____________,CNRT_MEM_TRANS_DIR_DEV2Host);
    cnrtConvertHalfToFloatArray(_____________________);
    //free data
    cnrtDestroy();
    ...
    return 0;
}
```

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

```
//filename: main.cpp
#include <math.h>
#include <time.h>
#include "stdio.h"
#include <stdlib.h>
#include <sys/time.h>

#define DATA_COUNT 32768
#define POW_COUNT //some num
int MLUPowerDifferenceOp(float* input1,float* input2, int pow, float*output, int dims_a)
    ;

int main() {
  //define
  ...
  FILE* f_input_x = fopen("./data/in_x.txt", "r");
  FILE* f_input_y = fopen("./data/in_y.txt", "r");
  FILE* f_output_data = fopen("./data/out.txt", "r");
  struct timeval tpend, tpstart;
  //load data
  ...
  //compute
  gettimeofday(&tpstart, NULL);
  MLUPowerDifferenceOp(input_x,input_y,POW_COUNT,output_data,DATA_COUNT);
  gettimeofday(&tpend, NULL);
  time_use = 1000000 * (tpend.tv_sec - tpstart.tv_sec)+ tpend.tv_usec - tpstart.tv_usec;
  for(int i = 0; i < DATA_COUNT;++i)
  {
     err +=fabs(output_data_cpu[i] - output_data[i]) ;
     cpu_sum +=fabs(output_data_cpu[i]);
  }
  printf("err rate = %0.4f%%\n", err*100.0/cpu_sum);
  return 0;
}
```

图 5.12　Main 程序示例代码

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

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

```
//filename: plugin_power_difference_op.cc

//cnmlCreatePluginPowerDifferenceOp
cnmlStatus_t cnmlCreatePluginPowerDifferenceOp(
  cnmlBaseOp_t *op,
  cnmlTensor_t* input_tensors,
  int power,
  cnmlTensor_t* output_tensors,
  int len
) {
  void** InterfacePtr;
  InterfacePtr = reinterpret_cast<void**>(&PowerDifferenceKernel);
  // Passing param
  cnrtKernelParamsBuffer_t params;
  cnrtGetKernelParamsBuffer(&params);
  cnrtKernelParamsBufferMarkInput(params);    // input 0
  cnrtKernelParamsBufferMarkInput(params);    // input 1
  cnrtKernelParamsBufferAddParam(params, &power, sizeof(int));
  cnrtKernelParamsBufferMarkOutput(params);   // output 0
  cnrtKernelParamsBufferAddParam(params, &len, sizeof(int));
  cnmlCreatePluginOp(op,
                     "PowerDifference",
                     InterfacePtr,
                     params,
                     input_tensors,
                     2,
                     output_tensors,
                     1,
                     nullptr,
                     0);
  cnrtDestroyKernelParamsBuffer(params);
  return CNML_STATUS_SUCCESS;
}
// cnmlComputePluginPowerDifferenceOpForward
cnmlStatus_t cnmlComputePluginPowerDifferenceOpForward(
  cnmlBaseOp_t op,
  void **inputs,
  void **outputs,
  cnrtQueue_t queue
) {
  cnmlComputePluginOpForward_V4(op,
                                nullptr,
                                inputs,
                                2,
                                nullptr,
                                outputs,
                                1,
                                queue,
                                nullptr);
  return CNML_STATUS_SUCCESS;
}
```

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。

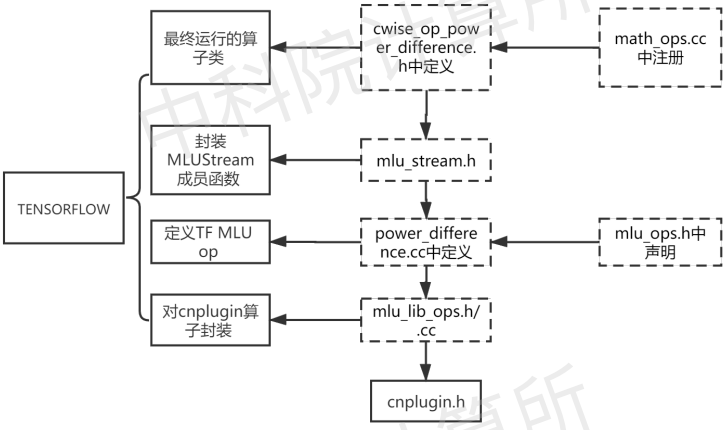

图 5.15　TensorFlow 集成流程图

• **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 cnmlCreatePluginPowerDifferenceOp 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/c
中。具体如图 5.17所示。

```
##tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.h
tensorflow::Status CreatePowerDifferenceOp(MLUBaseOp** op, MLUTensor* input1, MLUTensor*
    input2, int input3, MLUTensor* output, int len);
tensorflow::Status ComputePowerDifferenceOp(MLUBaseOp* op, MLUCnrtQueue* queue, void*
    input1, void* input2, void* output);

##tensorflow/stream_executor/mlu/mlu_api/lib_ops/mlu_lib_ops.cc
tensorflow::Status CreatePowerDifferenceOp(MLUBaseOp** op, MLUTensor* input1, MLUTensor*
    input2, int input3, MLUTensor* output, int len) {
  MLUTensor* inputs_ptr[2] = {input1, input2};
  MLUTensor* outputs_ptr[1] = {output};
  CNML_RETURN_STATUS(cnmlCreatePluginPowerDifferenceOp(op, inputs_ptr, input3,
    outputs_ptr, len));
}

tensorflow::Status ComputePowerDifferenceOp(MLUBaseOp* op, MLUCnrtQueue* queue, void*
    input1, void* input2, void* output) {
  void* inputs_ptr[2] = {input1, input2};
  void* outputs_ptr[1] = {output};
  CNML_RETURN_STATUS(cnmlComputePluginPowerDifferenceOpForward(op, inputs_ptr,
    outputs_ptr, queue));
}

###tensorflow/stream_executor/mlu/mlu_api/ops/mlu_ops.h 中添加声明
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。

```
//filename: tensorflow/stream_executor/mlu/mlu_api/ops/power_difference.cc
Status MLUPowerDifference::CreateMLUOp(std::vector<MLUTensor *> &inputs, std::vector<
    MLUTensor *> &outputs, void *param) {
  TF_PARAMS_CHECK(inputs.size() > 1, "Missing input");
  TF_PARAMS_CHECK(outputs.size() > 0, "Missing output");
  MLUBaseOp *power_difference_op_ptr = nullptr;
  MLUTensor *input1 = inputs.at(0);
  MLUTensor *input2 = inputs.at(1);
  int power_c = *((int*)param);
  MLUTensor *output = outputs.at(0);
  ...
  TF_STATUS_CHECK(lib::CreatePowerDifferenceOp(_______________));
  ...
}
Status MLUPowerDifference::Compute(const std::vector<void *> &inputs, const std::vector<
    void *> &outputs, cnrtQueue_t queue) {
  ...
  TF_STATUS_CHECK(lib::ComputePowerDifferenceOp(_______________));
  ...
}
```

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

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

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

```
//filename: tensorflow/core/kernels/cwise_op_power_difference_mlu.h
class MLUPowerDifferenceOp : public MLUOpKernel {
 public:
  explicit MLUPowerDifferenceOp(OpKernelConstruction* ctx) :
          MLUOpKernel(ctx) {}
  void ComputeOnMLU(OpKernelContext* ctx) override {
    //输入数据处理与条件判断
    ________________________________________________
    //Stream调用PowerDifference接口
    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。

```python
#power_difference_test_bcl.py
import numpy as np
import os
import time
#使用以下环境变量控制单算子的执行方式
os.environ['MLU_VISIBLE_DeviceS']="0"
os.environ['TF_CPP_MIN_MLU_LOG_LEVEL']="1"
import tensorflow as tf
np.set_printoptions(suppress=True)

def power_difference_op(input_x,input_y,input_pow):
    with tf.Session() as sess:
      x = tf.placeholder(tf.float32, name='x')
      y = tf.placeholder(tf.float32, name='y')
      pow_ = tf.placeholder(tf.float32, name='pow')
      z = tf.power_difference(x,y,pow_)
      return sess.run(z, feed_dict = {x: input_x, y: input_y, pow_: input_pow})

def main():
    start = time.time()
    input_x = np.loadtxt("./data/in_x.txt",delimiter=',')
    input_y = np.loadtxt("./data/in_y.txt")
    input_pow = np.loadtxt("./data/in_z.txt")
    output = np.loadtxt("./data/out.txt")
    end = time.time()
    print("load data cost "+ str((end-start)*1000) + "ms" )
    start = time.time()
    res = power_difference_op(input_x, input_y, input_pow)
    end = time.time()
    print("comput op cost "+ str((end-start)*1000) + "ms" )
    err = sum(abs(res - output))/sum(output)
    print("err rate= "+ str(err*100))

if __name__ == '__main__':
    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,

```cpp
//filename: src/style_transfer.cpp
#include "style_transfer.h"
#include <math.h>
#include <time.h>
#include "stdio.h"
#include <stdlib.h>
#include <sys/time.h>

int main(int argc, char** argv){
    // parse args
    std::string file_list = "/path/to/images/" + std::string(argv[1]) + ".jpg";
    std::string offline_model = "/path/to/models/offline_models/" + std::string(argv[2])
      + ".cambricon";

    //creat data
    DataTransfer* DataT =(DataTransfer*) new DataTransfer();
    DataT->image_name = argv[1];
    DataT->model_name = argv[2];
    //process image
    DataProvider *image = new DataProvider(file_list);
    image->run(DataT);

    //running inference
    Inference *infer = new Inference(offline_model);
    infer->run(DataT);

    //postprocess image
    PostProcessor *post_process = new PostProcessor();
    post_process->run(DataT);

    delete DataT;
    DataT = NULL;
    delete image;
    image = NULL;
    delete infer;
    infer = NULL;
    delete post_process;
    post_process = NULL;
}
```

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

```cpp
//filename: data_provider.cpp
#include "data_provider.h"

namespace StyleTransfer{
DataProvider :: DataProvider(std::string file_list_){
    ...
    set_mean();
}
void DataProvider :: set_mean(){
    float mean_value[3] = {
        0.0,
        0.0,
        0.0,
    };
    cv::Mat mean(256, 256, CV_32FC3, cv::Scalar(mean_value[0], mean_value[1], mean_value[2]));
    mean_ = mean;
}
bool DataProvider :: get_image_file(){
    image_list.push_back(file_list);
    return true;
}
cv::Mat DataProvider :: convert_color_space(std::string file_path){
    cv::Mat sample;
    cv::Mat img = cv::imread(file_path, -1);
    ...
    return sample;
}
cv::Mat DataProvider :: resize_image(const cv::Mat& source){
    cv::Mat sample_resized;
    cv::Mat sample;
    ...
    return sample_resized;
}
cv::Mat DataProvider :: convert_float(cv::Mat img){
    cv::Mat float_img;
    ...
    return float_img;
}
cv::Mat DataProvider :: subtract_mean(cv::Mat float_image){...}
void DataProvider :: split_image(DataTransfer* DataT){...}
DataProvider :: ~DataProvider(){}

void DataProvider :: run(DataTransfer* DataT){
    for(int i = 0; i < batch_size; i++){
        get_image_file();
        std::string img_path= image_list[i];
        cv::Mat img_colored = convert_color_space(img_path);
        cv::Mat img_resized = resize_image(img_colored);
        cv::Mat img_floated = convert_float(img_resized);
        DataT->image_processed.push_back(img_floated);
    }
    split_image(DataT);
}
}
```

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。

```
//filename: inference.cpp
#include "inference.h"
#include "cnrt.h"
...
namespace StyleTransfer{
Inference :: Inference(std::string offline_model){
    offline_model_ = offline_model;
}
void Inference :: run(DataTransfer* DataT){
    // load model
    // load extract function
    // prepare data on cpu
    // allocate I/O data memory on DLP
    // prepare input buffer
    // prepare output buffer
    // setup runtime ctx
    // bind Device
    // compute offline
    // free memory spac
}
} // 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 PowerDifferenece 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。

```cpp
//filename: post_processor.cpp
#include "post_processor.h"

namespace StyleTransfer{

PostProcessor :: PostProcessor(){
    std::cout << "PostProcessor constructor" << std::endl;
}

void PostProcessor :: save_image(DataTransfer* DataT){

    std::vector<cv::Mat> mRGB(3);
    for(int i = 0; i < 3; i++){
        cv::Mat img(256, 256, CV_32FC1, DataT->output_data + 256 * 256 * i);
        mRGB[i] = img;
    }
    cv::Mat im(256, 256, CV_8UC3);
    cv::merge(mRGB,im);

    std::string file_name = DataT->image_name + std::string("_") + DataT->model_name + ".jpg";
    cv::imwrite(file_name, im);
    std::cout << "style transfer result file: " << file_name << std::endl;
}

PostProcessor :: ~PostProcessor(){
    std::cout << "PostProcessor destructor" << std::endl;
}

void PostProcessor :: run(DataTransfer* DataT){
    save_image(DataT);
}

} // 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?