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、全连接等操作会被大量、重复的使用，而编程框架正是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。TensorFlow 具有较好的跨平台特性，可以运行在如 CPU、GPU、深度学习处理器等多种异构平台上。同时，TensorFlow 中提供了一系列高性能的 API，能够高效的实现深度学习算法。

本章首先以 VGG19 为例，介绍如何在 CPU 平台或 DLP 平台上使用 TensorFlow API 实现图像分类；在此基础上，介绍如何使用 TensorFlow 实现在 CPU 平台或 DLP 平台上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；随后，介绍了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以提高处理速度。

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，能够使用 TensorFlow 编程框架实现利用 VGG19 网络对输入图像进行分类，并在深度学习处理器 DLP 上实现图像分类。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 中几种常用数据结构的使用方法；
3. 掌握 TensorFlow 中几种常用 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。

本实验预计用时：1 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，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、深度学习处理器等异构硬件平台上的部署，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 VGG19[3] 为例，对每个卷积层来说，首先输入与卷积核（也称为权重）做卷积运算[8]，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，可以更方便的实现这些操作。

通常情况下，权重和偏置都是待训练的参数，因此在 TensorFlow 中使用变量来表示这两类数据。变量是计算图中的一种有状态节点，用于在多次执行同一计算图时存储并更新的指定张量，常用来表示机器学习或深度学习算法中的模型参数。作为有状态节点，变量的输出由输入、节点操作和节点内部已保存的状态值共同作用。

使用 TensorFlow 实现 VGG19 所需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。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[9]；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[10] 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 中用计算图来表示深度学习算法的网络拓扑结构。在做深度学习训练时，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。为了解决该问题，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；以及 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

神经网络的模型需要量化之后并存为.pb 格式的文件，才可以在 TensorFlow 框架下将神经网络运行在 DLP 平台。本实验平台提供了 TensorFlow 编程框架下的 fppb_to_intpb 量化工具，可以将 float32 型的模型文件量化为 int8 或 int16 型的模型文件。

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：

[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"NIIWC" 或"NCIIW"。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[11]。
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。

```
[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
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，目前只能量化 Conv、FC 和 LRN
channel_quantization = False           ; 是否使用分通道量化，目前不支持为 True

[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
```

```
27 batch_size = 1                    ; 每次运行的 batch_size
```

描写该参数配置文件时，需注意以下几点：

1. 关于 color_mode：

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

2. 关于 activation_quantization_alg 和 weight_quantization_alg：

   threshold_search 阈值搜索模式用于处理存在异常值的待量化数据集，该模式能够过滤部分异常值，重新计算出数据集的最值，用新最值来计算数据集的量化参数，从而提高数据集整体的量化质量。但是对于不存在异常值，数据分布紧凑的情况，不建议使用该算法，比如网络权值的量化。

3. 关于 device_mode:

   mlu：将输出 pb 的所有节点的 device 设置为 MLU。

   clean: 将输出 pb 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 一样的设备指定（在 int_op_list 中的算子除外）。

4. 关于 use_convfirst：

   convfirst 是一个优化，可以提升网络的性能，若一个网络要使用 convfirst，要满足以下几个条件：

   (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 ini 文件中的 [preprocess] 下面定义 mean、std 和 color_mode。

4.1.3 实验环境

1. 硬件环境：个人 PC、DLP 云平台

2. 软件环境：TensorFlow 1.14

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，实现平台包括 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几个步骤：

4.1.5.1 读取一张图像作为输入张量像

首先读入待进行计算的图像。在本实验中，利用 scipy.misc 模块内置的函数读入输入图像，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。

```
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

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。

```
def _conv_layer(input, weights, bias):
    # TODO: 定义卷积层的操作步骤，input 为输入张量，weights 为权重参数，bias 为偏置参数，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以及层命名，逐层定义需要执行的计算操作，每一层的输出作为下一层的输入。程序示例如下所示。

```
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、偏置加数，计算卷积结果 current
            ---------------
        elif name[:4] == 'relu':
        # TODO: 执行 ReLU 计算，计算结果存入 current
            ---------------
        # TODO: 完成其余层的定义，计算结果存入 current
        ---------------

        net[name] = current


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

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

```


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4.3所示。

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 框架上，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.pb 格式的量化过的深度学习模型文件，因此首先需要将模型文件保存为.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 格式的模型文件。此外，需要在程序中设置 DLP 的运行参数，这部分可以通过 config.mlu_options 进行配置。最后，编译运行 Python 程序来得到图像分类结果。

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 4.4所示。

2. **模型量化**

```
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)

        np.save('standard_conv1_1', preds['conv1_1'])
        np.save('standard_pool1', preds['pool1'])
        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("models/vgg19.pb", "wb") as f:
        f.write(output_graph_def.SerializeToString())
        print("**** Save Frozen Graph Done ****")
```

图 4.3　利用 VGG19 网络实现图像分类

```
import numpy as np
import struct
import os
import scipy.io
import time
import tensorflow as tf
from tensorflow.python.framework import graph_util #用于将模型保存为 pb 文件

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 4.4　将模型文件保存为 pb 格式

生成的 vgg19.pb 模型需要经过量化后才可以在 DLP 上运行，所以先要将原始的 float32 数据类型的 pb 模型量化成为 int 类型。在 vgg19/fppb_to_intpb 目录下运行下述命令，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。

```
import numpy as np
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)
```

```
50
51              prob = preds[0]
52              top1 = np.argmax(prob)
53
54              print('Classification result: id = %d, prob = %f' % (top1, prob[top1]))
```

4. **在 DLP 平台上实现图像分类**

在 DLP 平台上运行如下述命令，使用 VGG19 网络实现图像分类。

```
1       python main_exp_4_1.py
2
```

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、池化层操作的过程。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。

本部分得分占本章总得分的 30%。

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 实时风格迁移预测

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 定义一个完整的网络结构的方法；

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，在 CPU 上掌握使用 TensorFlow 进行神经网络预测的方法；

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 上使用 TensorFlow 对模型进行量化并推理的能力；

本实验预计用时：6 小时。

4.2.2 背景介绍

4.2.2.1 算法介绍

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，都需要通过对随机噪声图像的多次迭代训练得到风格迁移后的输出，耗时较长，实时性较差。因此，Johnson 等[12] 又提出了一种实时、快速的图像实时风格迁移算法。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。

图像转换网络的结构如图 4.5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）[13] 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。

表 4.3 图像转换网络中使用的结构参数

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

残差块

图像转换网络中包含了五个残差块，残差块的基本结构如图 4.6所示，与常规的卷积神经网络相比，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，都符合正态分布，因此输入和输出可

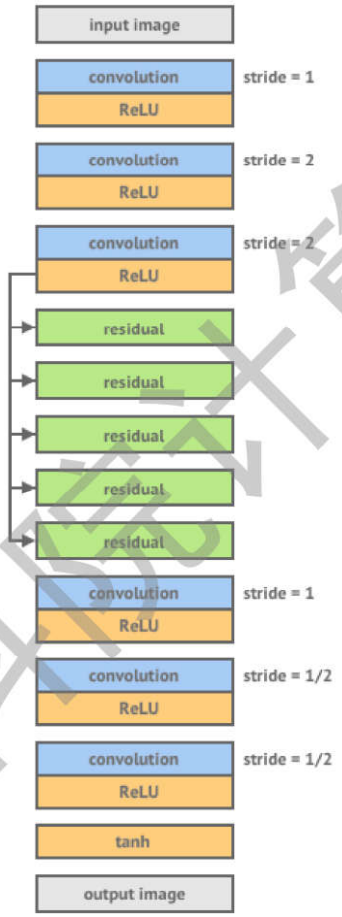

图 4.5　图像转换网络的网络结构

以做减法，如图 4.6中 $F(x) = H(x) - x$。而且从卷积和批归一化后得到的每层的输出响应的均方差来看，残差网络的响应小于常规网络[14]。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。

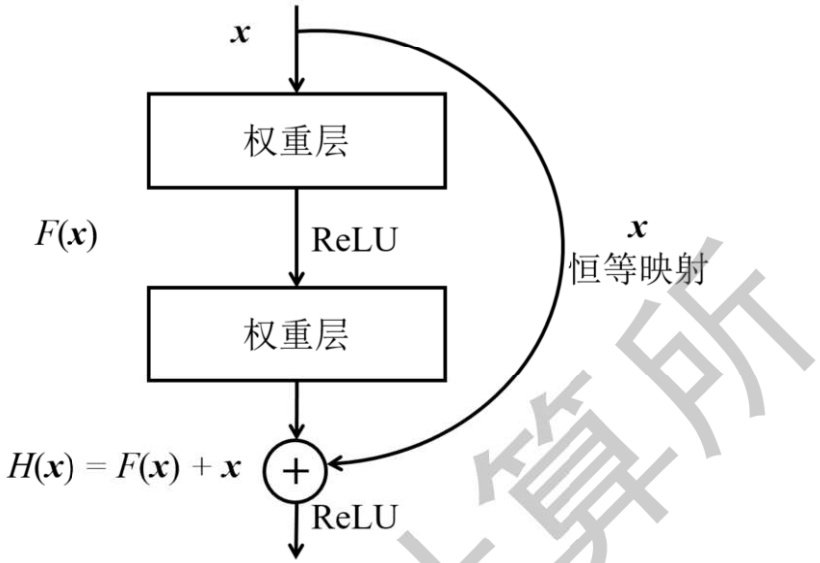

图 4.6　残差块结构

残差块的基本单元（基本块）的处理过程为：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 $F(\boldsymbol{x})$，再加上 $\boldsymbol{x}$ 得到输出 $H(\boldsymbol{x}) = F(\boldsymbol{x}) + \boldsymbol{x}$，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 $F(\boldsymbol{x})$ 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。

转置卷积

转置卷积[15] 又可以称为小数步长卷积，图 4.7是一个转置卷积的示例。下面以一个例子来说明转置卷积的计算步骤。假设输入矩阵 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。

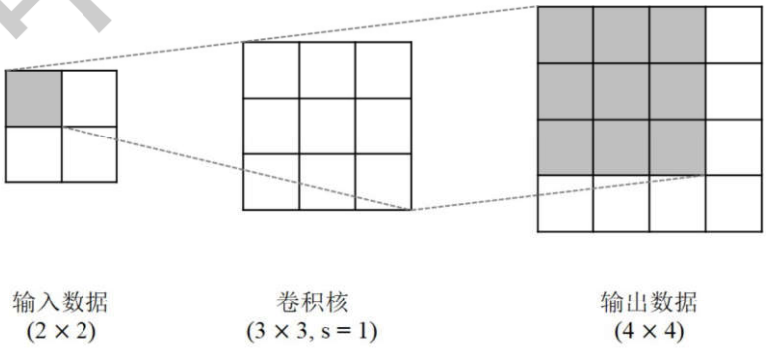

图 4.7　转置卷积

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 $\boldsymbol{W}$：

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

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

其中，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，其计算方法见公式 (4.1)。

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

其中，x_{tijk} 表示一个批次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 表示其在图像批次中的序号，i 表示特征通道序号。批归一化方法是在每个批次上分别对 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，所以对整个批次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[16] 。不同于批归一化，该方法使用表达式 (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。

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

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。

4.2.2.2 DLP 软件环境介绍

1. 整体环境

DLP 硬件的整体软件环境如图 4.8所示。整体包括 6 个部分：编程框架、高性能库、编程语言及编译器、运行时库及驱动、开发工具包及领域专用开发包等。上层的智能应用可以通过两种方式来运行：在线方式和离线方式。其中，在线方式直接用各种编程框架（如 TensorFlow、PyTorch、MXNet 和 Caffe 等）间接通过调用高性能库及运行时库来运行。离线方式通过直接调用运行时库，运行前述过程生成的特定格式网络模型，减少软件环境的中间开销，提升运行效率。以下重点介绍高性能库、运行时库以及开发工具包等。

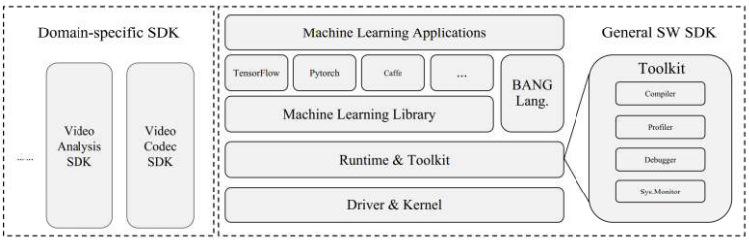

图 4.8　DLP 硬件的软件环境

2. 运行时库（CNRT）

DLP 的运行时库 CNRT 提供了面向 DLP 设备的用户层接口，用于完成包括设备管理、内存管理、任务管理等功能。运行时库作为 DLP 软件环境的底层支撑，其他应用层软件的运行都需要调用 CNRT 接口。CNRT 包括的主要功能如表 4.4所示。

表 4.4　DLP 运行时库功能介绍

功能模块	具体描述
设备管理	设备初始化、查询、指定等
内存管理	内存分配、释放、拷贝等
队列管理	队列（Queue）创建、销毁、同步等
通知管理	通知（Notifier）创建、销毁、同步等
任务管理	支持任务异步、并发、调度等

下述示例程序介绍了 mlp 算子的离线模型加载及计算过程，如图 4.10

完成 CNRT 离线代码编写后，还需要按照图 4.11所示指令进行编译和执行，得到图 4.12所示的结果。

3. 高性能库（CNML）

DLP 硬件的高性能库 CNML 提供了一套高效、通用、可扩展的编程接口，用于在 DLP 上加速各种智能算法。用户可以直接调用 CNML 中大量已优化的算子接口来实现其应用，同时可以根据自己的需求扩展已有算子。CNML 具有以下主要特性：

- 支持丰富的基本算子。已经支持的主要基本算子包括：常见神经网络运算（如卷积、池化和批规范化等），矩阵、向量及标量算子，循环神经网络算子（LSTM 和 RNN 等）等。
- 支持基本算子的融合。融合后的算子在编译期采用内存复用、片上存储管理和多核架构自适应等系列优化手段，显著提升融合算子的整体性能。
- 支持离线模型的生成。可以将编译好的算子（基本/融合）序列化到模型文件（离线模型）。该离线模型可以脱离编程框架和高性能库，采用底层运行时库接口直接运行。由于脱离了上次软件栈，离线模型的执行具有更好的性能和通用性。

除了提供针对特定算子预先优化好的高效实现外，CNML 中进一步提供了方便用户通过智能编程语言对高性能库算子进行扩展的插件 API（PluginOp）。通过 PluginOp，用户可以将自己用智能编程语言编写的算子“插入”到 CNML 高性能库中，与高性能库中原有算子的执行逻辑协同起来，支持包括算子融合和离线模型生成等多种功能。

下面将阐述如何使用 CNML 和 CNRT 库创建一个简单的 add 算子的过程。示例代码如下图 4.14所示，其主要分为 8 个步骤。

- 创建 CPU 端的张量，并准备 CPU 端的数据。
- 创建 MLU 端的张量, 利用 MLU 端的张量做为输入创建 AddOp。
- 编译。

- 将 CPU 端的数据拷入到 MLU 端。
- 执行 MLU 端的计算过程。
- 将 MLU 端的结果拷　到 CPU 端。
- 计算结果存放在文件中。
- 释放 MLU 和 CPU 端的资源。

完成 CNML 示例代码编写后，还需要按照图 4.15所示指令进行编译和执行。

4. 开发工具包

除了上述基本软件模块外，DLP 还提供了多种工具方便用户进行状态监测及性能调优，典型的如应用级性能剖析工具、系统级性能监控工具和调试器等。以下重点介绍本节实验中用到的应用级性能剖析工具和系统级性能监控工具。

应用级性能剖析工具（CNPerf）以性能事件为基础，可以精确地获得用户程序中每个函数的执行细节信息。如：函数调用栈信息；用户程序或部分依赖库中函数的执行时间；Host 侧和 DLP 侧的内存占用开销；高性能库中算子的计算效率及 DDR 访存带宽；用户自定义 Kernel 函数的实际执行时间等。

CNPerf 命令行主要支持以下几个命令:

- record: 记录性能数据并保存到数据文件中。数据文件默认保存在 dltrace_data 文件夹中，该命令可以使用相关参数改变该数据文件所在的默认路径。
- report: 在终端上显示目标程序的总体信息。
- replay: 在终端上显示所有日志信息。
- kernel: 在终端上展示更多 pmu 数据。
- show: 不记录日志，直接将数据打印到终端。
- monitor: 提供旁路指令，查看 Jpu/Vpu 利用率，以及 Vpu 读写的带宽和累计值。
- info: 显示本次 record 运行环境相关的信息。
- 显示所有命令的帮助信息。用戶执行该命令可以查看 CNPerf 支持的命令及其使用方法。

使用者可直接在/usr/local/neuware/bin 下执行 cnperf，或通过将上述目录添加至 PATH 环境变量中，即可在任意位置执行。

下面以命令行使用方法为例，简要介绍如何使用 cnperf 进行性能分析。生成并查看性能数据的步骤如下所示：

- 使用 record 命令生成性能数据，并保存到数据文件中。数据文件默认保存到生成的 dltrace_data 文件夹下。

- 以 dltrace_data 文件夹下的数据文件为输入，执行 report 命令查看性能数据信息。
- 以 dltrace_data 文件夹下的数据文件为输入，执行 replay 命令显示所有监测日志信息。
- 以 dltrace_data 文件夹下的数据文件为输入，执行 kernel 命令显示所有监测 pmu 性能信息。
- 使用 show 命令生成性能数据，不记录日志，直接在终端显示所有监测的性能信息。
- 使用 monitor 命令主要是用来提供旁路指令，查看 Jpu/Vpu 利用率，以及 Vpu 读写的带宽和累计值。

注意，CNPerf 跟踪的目标程序需要加-pg 参数编译，不加-pg 编译的目标程序 CNPerf 无法跟踪。举例如下: gcc test.c –pg –o test

下面对 test 可执行文件进行性能分析。

a. 通过指令跟踪 test 可执行文件：cnperf-cli record test。执行完毕后该目录会有 dltrace_data 文件夹生成。

b. 进入放置 dltrace_data 的目录，使用 report 命令查看 dltrace_data 数据信息：cnperf-cli report。得到如图 4.17所示信息，包含被跟踪程序的线程号、Function 执行时间、调用函数次数、kernel 的计算效率、kernel 执行时计算单元对设备内存的读写数量与总带宽等信息。

c. 此外还可以分别执行：cnperf-cli replay；cnperf-cli kernel；cnperf-cli monitor；cnperf-cli info 等指令获取相应的信息。

系统级性能监控工具（CNMon）主要通过利用驱动读取寄存器的方式来搜集硬件的静态和动态执行信息。在 DLP 硬件上可以采集的信息包括：硬件设备型号、驱动版本号、设备利用率、物理内存总量、虚拟内存总量、进程内存使用量、板卡功耗及温度、PCIe 信息等。

在正确安装 neuware-driver 后，即可在终端使用 cnmon 指令，得到如图 **??**的界面。分别包含：板卡号（Card）、板卡名称（Name）、 扇转速比（Fan）、功率和峰值功率（Pwr）、驱动版本（Driver）、利用率（Util）、虚拟内存使用情况（vMemory-Usage）、是否为虚拟机（VF）、固件版本（Firmware）、板卡温度（Temp）、cndev 是否初始化（Inited）、物理内存使用情况（Memory-Usage）、Ecc 报错数量统计（Ecc-Error）

4.2.2.3 模型量化介绍

由于 DLP 硬件支持如 INT8/INT16 等定点数据类型，可以用于对网络模型的处理进行加速。这里对量化的原理和具体方法展开介绍。

目前深度学习网络的权值、偏置、激活值等信息通常会使用 Float32 来表示，但当网络较大、网络层数较深时，网络参数也会随之增多，那么计算量以及访存空间无疑会面临巨大的压力。模型量化是解决此问题的方法之一，使用低精度的数据推理可以确保精度在可接受范围内损失的同时提升计算速度。

定点量化（以 INT8 为例）的实质是用一组共享指数位的定点数来表示一组浮点数。共享指数确定了二进制小数的小数点位置。将 FLOAT32 量化成 INT8 后，存储空间可以减少

为原来的 1/4。在神经网络中，权值、激活会集中在一个小范围内，因此可以进行量化，并且量化后对神经网络的精度不会产生很大的影响。量化后的网络不仅可以减少模型参数占用空间的大小，还可以大大加快神经网络的推理和训练速度。

模型量化的基本流程：首先准备所需的校准数据，该校准数据集最佳选择应为验证数据集的子集；在校准数据集运行 FP32 推理，针对每一层网络收集激活值的直方图并获取一组具有不同阈值的 8 位表示法，根据不同的阈值生成不同的量化分布，计算每个 INT8 分布与原 FP32 分布的量化误差，这个指标是用来衡量最优与次优之间的差异程度，量化误差越小说明 INT8 编码后的信息损失越少，此时的值即为对应的最佳比例因子：scale 值。

目前量化的方式支持在线和离线量化，对于 DLP 而言，主要以离线量化为主，用户通过调用相应的接口，设置具体的量化参数值即可完成。

下述公式中各符号表示如表 4.5所示：

表 4.5　量化符号与含义

数学符号	含义
r_x	需要定点量化的实数
q_x	定点量化后的整数
$position$	小数点的位置
$scale$	缩放系数
$offset$	偏移量
$round$	四舍五入取整

离线量化有以下两种方法：

1. 对称的定点表示：

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

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

2. 有缩放系数、对称的定点表示：

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

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

上述两种方法都基于对称的定点表示，如下图 4.18所示，设 Z 为需要量化表示的数域中所有数的绝对值最大值 $max(r_x)$，则 A 需要包含 Z，且 Z 要大于 $A/2$；position 的计算和对称定点表示相同，其中 n 表示量化类型的位宽，例如 int8 的位宽为 8，int16 的位宽为 16。$position$ 计算表示如下：

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

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

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

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

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

量化中常见的两种处理方法：

1. 对普通权值量化，即整个权值或输入内的所有数据都用相同的量化参数。
2. 按通道对权值或者输入进行量化，即整个张量内的数据按照不同的 channel，每个 channel 内的数据用相同的参数，不同 chanel 使用的参数可能不同。

通道量化是对 filter 张量输出通道的量化，来提升算法的计算精度。使用通道量化，需要设置 position 和 scale 参数，或者设置 position 和 1/scale 参数。如果只有 1/scale 的值，CNML 提供 alpha 参数来设置 1/scale 的值，避免了 1/scale 的计算来获得 scale 的值，从而提升了计算精度。在计算开销上，仅在编译阶段会略有不同。默认 position 的值为 0，scale 值为 1，alpha 值为 1。如果使用默认值，则执行单通道量化。

4.2.3 实验环境

1. 硬件环境：个人 PC、DLP 云平台
2. 软件环境：TensorFlow 1.14

4.2.4 实验内容

构建图像转换网络，并加载预先训练好的模型参数。然后将输入图像读入到构建好的图像转换网络中，实时的输出风格迁移后的图像[17][①]。

由于 DLP 硬件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 定制版本的 TensorFlow 上运行，并和 CPU 版本的 TensorFlow 版本进行性能对比。具体实验内容主要包括：

1. **模型量化**：由于 DLP 硬件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 FP32 类型表示的 TensorFlow 模型量化为用 INT8 类型表示的 pb 模型。
2. **模型推理**：将 pb 模型通过 Python 接口在 DLP 上运行推理过程，并和第 5 章在 CPU 上运行推理的性能进行对比。

4.2.5 实验步骤

本实验主要包括以下几步：

4.2.5.1 CPU 上实时风格迁移预测的实现

为了在 CPU 上实现实时风格迁移预测，需要令输入的内容图像通过图像转换网络对应的 pb 模型文件，得到风格迁移后的输出图像。

[①]第4.2和4.3节的代码参考自：https://github.com/lengstrom/fast-style-transfer/。

1. 实时风格迁移预测方法定义

以下图中的代码为例说明实时风格迁移预测方法的定义。该函数定义在 stu_upload/evaluate_cpu.py 文件中。

```
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，则依次将该批次中输入图像文件路径下的 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
                    _____________
                delta_time = end - start
                print("Inference (MLU) processing time: %s" % delta_time)

```

2. 实时风格迁移训练主函数

以下面的代码为例说明实时风格迁移预测方法的定义。该函数同样定义在 stu_upload/evaluate_cpu.p 文件中。

```
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():
    #实时风格迁移预测主函数
    #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()

```

3. 执行实时风格迁移预测

在个人 PC 上运行如图 4.19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/路径下，输入的内容图像保存在 data/train2014_small/路径下，风格迁移后的图像保存在 out/路径下。

4.2.5.2 在 DLP 上实现实时风格迁移的预测

在 DLP 上实现实时风格迁移预测的实验步骤分为：模型量化和模型推理两部分。

1. 模型量化

   模型量化主要用于将原始的 FP32 类型表示的模型量化为 INT8 类型表示的模型。针对原始的 FP32 表示的模型，首先需要将原始模型的 Graph 进行改造，在其中每个 Conv2D 和 MatMul 操作前面插入两个转数节点（FakeQuantScaleFix8Gen），分别用于将输入和权值数据进行转换。风格迁移模型的 Graph 在改造前后部分结构对比如图4.20所示。通过运行改造好的 Graph，并将每个 Conv2D 和 MatMul 的输入和权值数据保存下来用于计算 input_scale 和 filter_scale。最后，将原始 Graph 中的 Conv2D 替换为 Iint8Conv2D，将 MatMul 替换为 Int8MatMul，将 LRN 替换为 Int8LRN。为了简化上述量化过程，DLP 软件环境提供了相应的模型量化工具，该工具的使用方法如图 4.21所示。其运行所需的配置文件如图 4.22所示，其可通过 DLP 提供的配置文件生成器生成，读者也可以选择手动创建。

2. 模型推理模型推理指通过 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理。为了使上层用户不感知迁移底层硬件的迁移，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 4.23所示，其中可以设置运行的核数、使用的数据类型以及是否要生成离线模型等信息。在配置完 DLP 硬件相关的参数后，推理时模型的算子可自动运行在 DLP 上。

   在运行完成后，统计 sess.run() 前后的运行时间，与第 5 章在 CPU 上的运行时间进行对比。

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预测的全过程，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预测的时间。
- 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，同时给出基于 DLP 硬件对图像进行实时风格迁移预测的时间并和 CPU 对比。

本部分得分占本章总得分的 30%。

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. 如果将图像转换网络中某一层权重参数的数据类型由浮点数转换成低位宽定点数，应该选择怎样的量化方法，才能使生成的结果基本不受影响？

4. 请采用性能剖析/监控等工具分析 DLP 推理的性能瓶颈。如何利用多核 DLP 架构提升整体的吞吐？

```
#include "cnrt.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int offline_test(const char *name) {
    // prepare model name
    char fname[100] = "../";
    strcat(fname, name);
    strcat(fname, ".dlp");
    // load model
    cnrtModel_t model;
    cnrtLoadModel(&model, fname);

    cnrtInit(0);
    unsigned dev_num;
    cnrtGetDeviceCount(&dev_num);
    if(dev_num == 0){
        exit(-1);
    }
    cnrtDev_t dev;
    cnrtGetDeviceHandle(&dev, 0);
    cnrtSetCurrentDevice(dev);

    // get model total memory
    int64_t totalMem;
    cnrtGetModelMemUsed(model, &totalMem);
    printf("total memory used: %ld Bytes\n", totalMem);
    // get model parallelism
    int model_parallelism;
    cnrtQueryModelParallelism(model, &model_parallelism);
    printf("model parallelism: %d.\n", model_parallelism);
    // load extract function
    cnrtFunction_t function;
    cnrtCreateFunction(&function);
    cnrtExtractFunction(&function, model, name);
    int inputNum, outputNum;
    int64_t *inputSizeS, *outputSizeS;
    cnrtGetInputDataSize(&inputSizeS, &inputNum, function);
    cnrtGetOutputDataSize(&outputSizeS, &outputNum, function);
    // prepare data on cpu
    void **inputCpuPtrS = (void **)malloc(inputNum * sizeof(void *));
    void **outputCpuPtrS = (void **)malloc(outputNum * sizeof(void *));
    // allocate I/O data memory on MLU
    void **inputMluPtrS = (void **)malloc(inputNum * sizeof(void *));
    void **outputMluPtrS = (void **)malloc(outputNum * sizeof(void *));
    // prepare input buffer
    for (int i = 0; i < inputNum; i++) {
        // converts data format when using new interface model
        inputCpuPtrS[i] = malloc(inputSizeS[i]);
        // malloc mlu memory
        cnrtMalloc(&(inputMluPtrS[i]), inputSizeS[i]);
        cnrtMemcpy(inputMluPtrS[i], inputCpuPtrS[i], inputSizeS[i],
     CNRT_MEM_TRANS_DIR_HOST2DEV);
    }
    // prepare output buffer
    for (int i = 0; i < outputNum; i++) {
        outputCpuPtrS[i] = malloc(outputSizeS[i]);
        // malloc mlu memory
        cnrtMalloc(&(outputMluPtrS[i]), outputSizeS[i]);
    }
```

```
    // prepare parameters for cnrtInvokeRuntimeContext
    void **param = (void **)malloc(sizeof(void *) * (inputNum + outputNum));
    for (int i = 0; i < inputNum; ++i) {
        param[i] = inputMluPtrS[i];
    }
    for (int i = 0; i < outputNum; ++i) {
        param[inputNum + i] = outputMluPtrS[i];
    }
    // setup runtime ctx
    cnrtRuntimeContext_t ctx;
    cnrtCreateRuntimeContext(&ctx, function, NULL);
    // bind device
    cnrtSetRuntimeContextDeviceId(ctx, 0);
    cnrtInitRuntimeContext(ctx, NULL);
    // compute offline
    cnrtQueue_t queue;
    cnrtRuntimeContextCreateQueue(ctx, &queue);
    // invoke
    cnrtInvokeRuntimeContext(ctx, param, queue, NULL);
    // sync
    cnrtSyncQueue(queue);
    // copy mlu result to cpu
    for (int i = 0; i < outputNum; i++) {
        cnrtMemcpy(outputCpuPtrS[i], outputMluPtrS[i], outputSizeS[i],
     CNRT_MEM_TRANS_DIR_DEV2HOST);
    }
    // free memory space
    for (int i = 0; i < inputNum; i++) {
        free(inputCpuPtrS[i]);
        cnrtFree(inputMluPtrS[i]);
    }
    for (int i = 0; i < outputNum; i++) {
        free(outputCpuPtrS[i]);
        cnrtFree(outputMluPtrS[i]);
    }
    free(inputCpuPtrS);
    free(outputCpuPtrS);
    free(param);
    cnrtDestroyQueue(queue);
    cnrtDestroyRuntimeContext(ctx);
    cnrtDestroyFunction(function);
    cnrtUnloadModel(model);
    cnrtDestroy();
    return 0;
}
int main() {
    printf("mlp offline test\n");
    offline_test("mlp");
    return 0;
}
```

图 4.10　CNRT 离线示例

```
export NEUWARE=/path/to/neuware
g++ -c cnrt_mlp.cpp -I/path/to/neuware/include
g++ cnrt_mlp.o -o cnrt_mlp -L /path/to/neuware/lib64 -lcnrt
```

图 4.11　CNRT 离线示例编译

```
./cnrt_mlp

mlp offline test
CNRT: 4.2.1 fa5e44c
total memory used: 29851424 Bytes
model parallelism: 1.
```

图 4.12　CNRT 离线示例输出

```
// define tensors: input, output
/*
 * op name: add
 * input_1 size: n x c x h x w
 * input_2 size: n x c x h x w
 * output size: n x c x h x w
 */
#include "cnml.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int add_test() {
    const cnmlCoreVersion_t coreVersion = CNML_MLU270;
    cnrtInit(0);
    unsigned dev_num;
    cnrtGetDeviceCount(&dev_num);
    if(dev_num == 0){
        exit(-1);
    }
    cnrtDev_t dev;
    cnrtGetDeviceHandle(&dev, 0);
    cnrtSetCurrentDevice(dev);

    // prepare data for pool
    const int dimNum = 4;
    const int n = 1, c = 32, h = 4, w = 4;
    const int coreNum = 4;
    // count input, filter, bias, output nums
    int input_count_1 = n * h * w * c;
    int input_count_2 = n * h * w * c;
    int output_count = n * h * w * c;
    float *input_cpu_ptr_1 = (float *)malloc(input_count_1 * sizeof(float));
    float *input_cpu_ptr_2 = (float *)malloc(input_count_2 * sizeof(float));
    float *output_cpu_ptr = (float *)malloc(output_count * sizeof(float));
    unsigned int seed = 123;
    for (int i = 0; i < input_count_1; i++) {
          input_cpu_ptr_1[i] = ((rand_r(&seed) % 100 / 100.0) - 0.5) / 2;
    }
    for (int i = 0; i < input_count_2; i++) {
    input_cpu_ptr_2[i] = (rand_r(&seed) % 100 / 100.0) - 0.5;
    }
    // set tensor shapes
    int input_shape_1[] = {n, c, h, w};
    int input_shape_2[] = {n, c, h, w};
    int output_shape[] = {n, c, h, w};
    // prepare input tensor 1
    cnmlTensor_t input_tensor_1 = NULL;
    cnmlCreateTensor_V2(&input_tensor_1, CNML_TENSOR);
    cnmlSetTensorShape_V2(input_tensor_1, dimNum, input_shape_1, NULL);
    cnmlSetTensorDataType(input_tensor_1, CNML_DATA_FLOAT32);
    // prepare input tensor 2
    cnmlTensor_t input_tensor_2 = NULL;
    cnmlCreateTensor_V2(&input_tensor_2, CNML_TENSOR);
    cnmlSetTensorShape_V2(input_tensor_2, dimNum, input_shape_2, NULL);
    cnmlSetTensorDataType(input_tensor_2, CNML_DATA_FLOAT32);
```

```
    // prepare output tensor
    cnmlTensor_t output_tensor = NULL;
    cnmlCreateTensor_V2(&output_tensor, CNML_TENSOR);
    cnmlSetTensorShape_V2(output_tensor, dimNum, output_shape, NULL);
    cnmlSetTensorDataType(output_tensor, CNML_DATA_FLOAT32);
    // create add operator
    cnmlBaseOp_t add_op;
    cnmlCreateAddOp(&add_op, input_tensor_1, input_tensor_2, output_tensor);
    // compile op
    cnmlSetBaseOpCoreVersion(add_op, coreVersion);
    cnmlSetBaseOpCorenum(add_op, coreNum);
    cnmlCompileBaseOp_V2(add_op);
    // mlu buffer ptr
    void *input_mlu_ptr_1 = NULL;
    void *input_mlu_ptr_2 = NULL;
    void *output_mlu_ptr = NULL;
    // malloc cnml tensor
    cnrtMalloc(&input_mlu_ptr_1, input_count_1 * sizeof(float));
    cnrtMalloc(&input_mlu_ptr_2, input_count_2 * sizeof(float));
    cnrtMalloc(&output_mlu_ptr, output_count * sizeof(float));
    // copy input to cnml buffer
    cnrtMemcpy(input_mlu_ptr_1, input_cpu_ptr_1, input_count_1 * sizeof(float),
     CNRT_MEM_TRANS_DIR_HOST2DEV);
    cnrtMemcpy(input_mlu_ptr_2, input_cpu_ptr_2, input_count_2 * sizeof(float),
     CNRT_MEM_TRANS_DIR_HOST2DEV);
    // set cnrt queue
    cnrtQueue_t queue;
    cnrtCreateQueue(&queue);
    cnmlComputeAddOpForward_V4(add_op, NULL, input_mlu_ptr_1, NULL, input_mlu_ptr_2,
     NULL, output_mlu_ptr, queue, NULL);
    //wait for computing task over cnrtSyncQueue(queue);
    //end of queue life cycle cnrtDestroyQueue(queue);
    // copy output to cpu
    cnrtMemcpy(output_cpu_ptr, output_mlu_ptr, output_count * sizeof(float),
     CNRT_MEM_TRANS_DIR_DEV2HOST);
    // dump mlu result to file mlu_output
    cnmlDumpTensor2File_V2("mlu_output", output_tensor, output_cpu_ptr, false);
    // delete op ptr
    cnmlDestroyBaseOp(&add_op);
    // delete cnml buffer
    cnrtFree(input_mlu_ptr_1);
    cnrtFree(input_mlu_ptr_2);
    cnrtFree(output_mlu_ptr);
    // delete cnml tensors
    cnmlDestroyTensor(&input_tensor_1);
    cnmlDestroyTensor(&input_tensor_2);
    cnmlDestroyTensor(&output_tensor);
    // delete pointers (including data pointers)
    free(input_cpu_ptr_1);
    free(input_cpu_ptr_2);
    free(output_cpu_ptr);
    return 0;
}
int main() {
    printf("cnml add test\n");
    add_test();
    return 0;
}
```

图 4.14　CNML 示例程序

```
#编译
export NEUWARE=/path/to/neuware
g++ -c cnml_demo.cpp -I/path/to/neuware/include
g++ cnml_demo.o -o cnml_demo -L /path/to/neuware/lib64 -lcnrt -lcnml
#执行
./cnml_demo
#输出
cnml add test
CNRT: 4.2.1 fa5e44c
```

图 4.15　CNML 示例程序编译

```
PID    Total time   Self time     Calls  Function
=====  ==========   =========   =======  ======================================
25150  918.722 us  918.722 us       228  cnrtInvokeFunction
       698.348 us  698.348 us       269  cnmlBindConstData
       666.434 us  666.434 us       228  cnrtCreateKernelHeapAllocInfo
       623.647 us  623.647 us       228  cnrtUpdateBarrierInstV3
       ...
         3.204 us    3.204 us         2  cnrtInvokeKernelRecordList_clear
         3.044 us    3.044 us         1  cnmlDestroyConvFirstOpParam
------  ----------  ----------  --------  --------------------------------------

Kernels Info:
  Pid  Duration   ComputeSpeed IOSpeed      IOCount  Function
=====  ========  ===========  ==========  =======  ================================
25150  138.00 us  7721 GOPS/s  4.973 GiB/s  736894  cnmlComputeConvFirstOpForward_V3[1]
       146.00 us  5137 GOPS/s  10.41 GiB/s 1631363  cnmlComputeBatchNormOpForward_V3[1]
       146.00 us  5137 GOPS/s  10.41 GiB/s 1631363  cnmlComputeScaleOpForward_V3[1]
       ...
       127.00 us  8366 GOPS/s  15.52 GiB/s  2116409  cnmlComputeMlpOpForward_V3[1]
        11.00 us  9500 GOPS/s  1.478 GiB/s  17454    cnmlComputeSoftmaxOpForward_V3[1]
-----  --------  -----------  ----------  -------  --------------------------------

Max MLU: 0 MB
DEVICE_TO_HOST size: 0MB
HOST_TO_DEVICE size: 42.7614MB speed: 0.00514178 GiB/s
```

图 4.16　CNPerf Report

```
+-----------------------------------------------------------------------------+
| CNMON 1.11.0                                                                |
+-------------------------------+----------------------+----------------------+
| Card  VF   Name     Firmware  | Inited       Driver  | Util        Ecc-Error |
| Fan   Temp     Pwr:Usage/Cap  |         Memory-Usage |         vMemory-Usage |
|===============================+======================+======================|
| 0     /    MLU270   v0.2.0    | On           v2.2.0  | 0%          0         |
| 19%   44C        22 W/ 150 W  |  2705 MiB/ 16384 MiB | 10240 MiB/1048576 MiB |
+-------------------------------+----------------------+----------------------+


+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  Card  VF  PID    Command Line                              MLU Memory Usage  |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

图 4.17　CNMon 监控工具

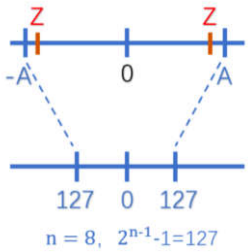

图 4.18　INT8 量化

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

图 4.19　执行实时风格迁移预测

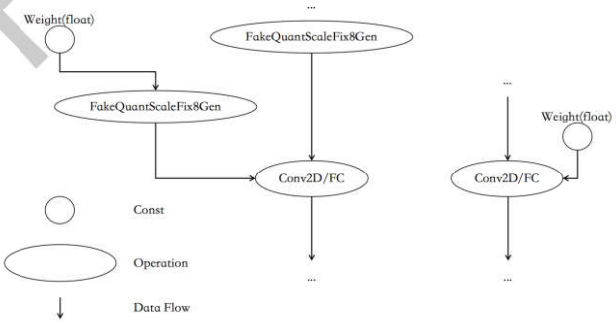

图 4.20　Graph 改造前后的部分结构对比

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf
import ...
import ConfigParser
from tensorflow.contrib import camb_quantize
def read_pb(input_model_name):
    ...
    return input_graph_def
def create_pb(output_graph_def, output_model_name):
    ...
    print("cpu_pb transform to mlu_int8_pb is finished")
def process_images_and_get_input_max_min(...):
    #Precess input images and get input max and min

if __name__ == '__main__':
    #set model_param, data_param, preprocess_param, config_param
    fixer = camb_quantize.QuantizeGraph(
                    input_graph_def = input_graph_def,
                    output_node_names = output_node_names,
                    use_convfirst = use_convfirst,
                    convfirst_params = convfirst_params,
                    device_mode = device_mode,
                    int8_op_list = int8_op_list)
    process_images_and_get_input_max_min(...,fixer,...)
    output_graph_def = fixer.rewrite_int8_graph()
    create_pb(output_graph_def, output_model_name)
```

图 4.21　通过 TensorFlow 接口进行 DLP 硬件上的 INT8 量化

```
#配置文件生成
[model]
original_models_path = style_transfer.pb
save_model_path = style_transfer_int8.pb
input_nodes = X_content
output_nodes = add_37

[data]
images_list = ./images_list
used_images_num = 6
batch_size = 1
iters = 6

[preprocess]
color_mode = rgb
mean = 123.68, 116.78, 103.94
std = 1
crop = 256, 256

[config]
int8_op_list = Conv, FC, LRN
use_convfirst = False
device_mode = mlu
```

图 4.22　DLP 硬件 INT8 量化的配置文件

```
import tensorflow as tf
#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
config.mlu_options.offline_model_name = "name.cambricon"
sess  = tf.Session(config = config, graph = graph)
```

图 4.23　用 DLP 进行模型推理时配置参数

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 实现实时风格迁移算法的训练。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；
4. 本实验预计用时：20 小时。

4.3.2 背景介绍

在第4.2小节中，介绍了使用 TensorFlow 实现实时风格迁移预测的方法。本部分将进一步介绍如何使用 TensorFlow 来实现实时风格迁移的训练。

除了第4.2小节中用于预测的图像转换网络，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算法流程如图 4.24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[18]，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，在损失函数中增加全变分正则化 (Total Variation Regularization)[19] 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，然后计算二者 L^2 范数的和。此外，将特征提取

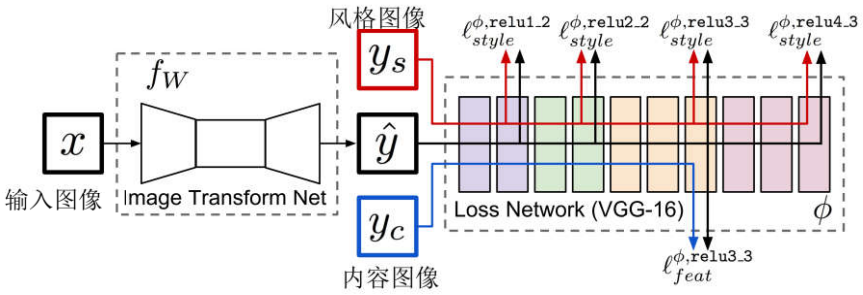

图 4.24　实时图像风格迁移算法的流程[12]

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，效果也会更好。VGG19 的网络结构与 VGG16 的区别如表 4.6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。

表 4.6　VGG19 与 VGG16 的区别[2]

ConvNet 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					

在训练图像转换网络的过程中，输入图像 (即内容图像)$\boldsymbol{x}$ 送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、风格图像 $\boldsymbol{y}_s$、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，且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

1. 硬件环境：个人 PC、DLP 云平台

2. 软件环境：TensorFlow 1.14

4.3.4 实验内容

构建如图 4.24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[17]，最终获得较好的训练效果。

4.3.5 实验步骤

4.3.5.1 读入输入图像

首先读入待进行计算的图像。在本实验中，利用 scipy.misc 模块内置的函数读入输入图像，并将图像处理成便于数值计算的 ndarray 类型。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中的 exp_4_2_fast_style_transfer_infer/src/utils.py 文件中，下面以图 4.25中的代码为例进行说明。

```
1   import scipy.misc
2   import numpy as np
3
4   def get_img(src, img_size = False):
5   #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
6   img = _____
7   _____
8
9   return img
10
```

图 4.25 读入输入图像

4.3.5.2 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 exp_4_2_fast_style_transfer_infer/src/transform.py 文件中。

1. 卷积层

   以图 4.26中的代码为例介绍图像转换网络中卷积层的定义方法。

2. 残差块

   以图 4.27中的代码为例介绍图像转换网络中残差块的定义方法。

```
    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 4.26　卷积层的定义

```
    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 4.27　残差块的定义

3. 转置卷积层

以图 4.28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。

```
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 4.28　转置卷积层的定义

4.3.5.3　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 4.5所示的图像转换网络模型。该部分代码定义在 exp_4_2_fast_style_transfer_infer/src/transform.py 文件中，下面以图 4.29中的代码为例来介绍。

```
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 4.29　创建图像转换网络模型

4.3.5.4　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。下面是定义特征提取网络的程序代码。该部分代码定义在 exp_4_2_fast_style_transfer_infer/src/vgg.py

文件中。

```
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
```

4.3.5.5 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 exp_4_2_fast_style_transfer_infer/src/optimize.py 文件中，同时会调用前几个步骤中实现的 vgg.py 及 transform.py 文件。

```
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)

    #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]
        #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
```

4.3.5.6 实时风格迁移训练的实现

在完成了特征提取网络以及损失函数的定义后，接下来需要创建优化器，定义模型训练方法，最后迭代地执行模型的训练过程。该部分代码定义在 exp_4_2_fast_style_transfer_infer/src/optimize.p 以及 exp_4_2_fast_style_transfer_infer/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 exp_4_2_fast_style_transfer_infe 文件中，同时会调用前几个步骤中实现的 vgg.py、transform.py 以及 utils.py 文件。

```
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]

        if slow:
            preds = tf.Variable(tf.random_normal(X_content.get_shape()) * 0.256)
            preds_pre = preds
        else:
            #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
            _____

        #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
        net = _____

```

```
        #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
        _____________
        #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
        _____________
        #TODO：初始化所有变量
        _____________
        import random
        uid = random.randint(1, 100)
        print("UID: %s" % uid)
        save_id = 0
        for epoch in range(epochs):
            num_examples = len(content_targets)
            iterations = 0
            while iterations * batch_size < num_examples:
                start_time = time.time()
                curr = iterations * batch_size
                step = curr + batch_size
                X_batch = np.zeros(batch_shape, dtype=np.float32)
                for j, img_p in enumerate(content_targets[curr:step]):
                    X_batch[j] = get_img(img_p, (256,256,3)).astype(np.float32)

                iterations += 1
                assert X_batch.shape[0] == batch_size

                feed_dict = {
                    X_content:X_batch
                }

                train_step.run(feed_dict=feed_dict)
                end_time = time.time()
                delta_time = end_time - start_time
                is_print_iter = int(iterations) % print_iterations == 0
                if slow:
                    is_print_iter = epoch % print_iterations == 0
                is_last = epoch == epochs - 1 and iterations * batch_size >= num_examples
                should_print = is_print_iter or is_last
                if should_print:
                    to_get = [style_loss, content_loss, loss, preds]
                    test_feed_dict = {
                       X_content:X_batch
                    }

                    tup = sess.run(to_get, feed_dict = test_feed_dict)
                    _style_loss,_content_loss,_loss,_preds = tup
                    losses = (_style_loss, _content_loss, _loss)
                    if slow:
                        _preds = vgg.unprocess(_preds)
                    else:
                        with tf.device('/CPU:0'):
                            #TODO：将模型参数保存到 savc_path，并将训练的次数 savc_id 作为后缀加入到模型名
字中
                            _____________
                    #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

以下面的代码来实现实时风格迁移训练的主函数。该函数定义在 exp_4_2_fast_style_transfer_infer/st

文件中，同时会调用前几个步骤中实现的 optimize.py、utils.py 以及 evaluate.py 文件[①]。

```
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'),
        "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 = [
```

[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 iteration，同时每隔 100 个 iteration 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。

```
        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

在实验环境中运行如下命令，实现实时风格迁移的训练。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 \
--batch-size 4

```

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 loss 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 loss 值逐渐减少。

[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。

- 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。

本部分得分占本章总得分的 40%。

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 是如何以计算图的方式完成对深度学习算法的处理；
3. 通过添加自定义的 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：6 学时

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。

本节要求实现的 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，最后再进行求幂运算。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，最简便的方式是使用 Python 根据已有的 Python 操作（Op）来构造新的 Op。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 Op。具体来说，是否采用 C++ 来实现 Op 有以下几条基本准则：

1. 使用现有的 Op 并加以组合成新的 Op 不易实现或无法实现；
2. 将新的 Op 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操作，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“Median Pooling”功能函数，其类似于“Max Pooling”运算，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操作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 CPU 算子，一般需要以下 5 个步骤：

1. 用 C++ 实现 Op 的具体功能。Op 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 和 DLP 等），可以有多个 Kernel 实现。
2. 在 C++ 文件中注册新 Op。Op 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 Op 如何执行。例如，Op 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 Op 的公共 API。默认的封装器是通过 Op 注册遵循特定规则生成的，用户可以直接使用。
4. 编写该 Op 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：安装有 TensorFlow-1.14 版本的 PC 或云平台设备。该实验仅会用到 CPU，因此亦可在读者自己的电脑或嵌入式设备中进行实验。
- 软件环境：TensorFlow-1.14 源码；Bazel 安装包。推荐读者使用 DLP 开源的 TensorFlow 源码进行实验。

4.4.4 实验内容

基于第 4 章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 4.30所示。主要流程包括：

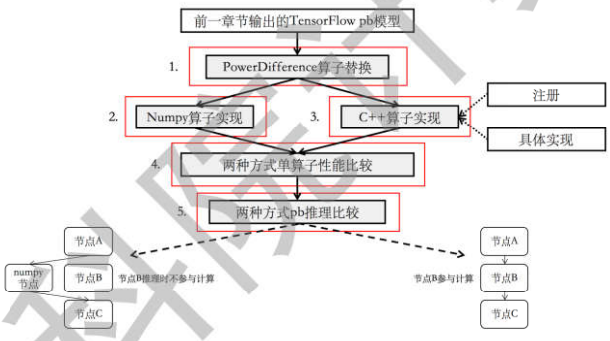

图 4.30　自定义算子实验流程图

1. 模型扩展：TensorFlow pb 模型节点的修改与扩展；
2. NumPy 实现：使用 NumPy 实现该算子的数学计算；
3. C++ 实现：使用 C++ 完成该算子的 CPU 实现并集成在框架中；
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 模型推理的性能差异。

最后一个步骤中包含 NumPy 算子和 C++ 算子的 pb 模型推理测试。对于 NumPy 算子需要将 PowerDifference 算子的输入直接输出到 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，仅需正常执行推理程序，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

在进行模型扩展之前，由于从前一章节得到的实时风格迁移模型为 TensorFlow 的 ckpt 模型，需要先将其转换为 frozen pb 模型后才能进行下面的实验。具体来说只需将图 4.31中的代码添加在前一章节 xx 部分，即可通过 session 实例将模型保存为新的格式。

```
  with tf.Session(...) as session:
    ...
    output_tensor = sess.graph.get_tensor_by_name('add_37:0')

    frozen_graph = tf.graph_util.convert_variables_to_constants(session,
            session.graph.as_graph_def(), output_tensor)

    export_file = os.path.join(FLAGS.output_dir, "name.pb")
    with tf.gfile.GFile(export_file, "wb") as f:
      f.write(frozen_graph.SerializeToString())
    tf.logging.info("**** Save Frozen Graph Done ****")
```

图 4.31 ckpt-pb 模型转换

模型扩展主要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 pb2pbtxt 工具（如图 4.32所示），将 pb 模型转换为可读的 pbtxt 格式。以 udnie.pb 模型为例，通过指令生成 udnie.pbtxt 文件：

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

```
import tensorflow as tf
from google.protobuf import text_format
from tensorflow.python.platform import gfile
import sys

if len(sys.argv) != 3:
    print("{} pb_path pbtxt_path".format(sys.argv[0]))
pb_path= sys.argv[1]
pbtxt_path = sys.argv[2]

def graphdef_to_pbtxt(filename,pbtxt_path):
    with gfile.FastGFile(filename,'rb') as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())
        tf.import_graph_def(graph_def, name='')
        tf.train.write_graph(graph_def, './', pbtxt_path, as_text=True)
    return
graphdef_to_pbtxt(pb_path,pbtxt_path)
```

图 4.32 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 4.33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 4.33　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 4.34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。

```
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 4.34　pbtxt 修改节点属性

4. **输出扩展模型：**采用 pbtxt2pb 工具（如图 4.35所示），将编辑后的 udnie.pbtxt 输出为扩展后的 pb 模型 udnie_power_diff.pb。

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 实现的程序如图 4.36所示。

```python
import tensorflow as tf
from google.protobuf import text_format
from tensorflow.python.platform import gfile
import sys

if len(sys.argv) != 3:
    print("{} pb_path pbtxt_path".format(sys.argv[0]))

pbtxt_path = sys.argv[1]
pb_path= sys.argv[2]

print(tf.__version__)
def pbtxt_to_graphdef(pbtxt_path,pb_path):
      with open(pbtxt_path, 'r') as f:
          graph_def = tf.GraphDef()
          file_content = f.read()
          text_format.Merge(file_content, graph_def)
          tf.import_graph_def(graph_def, name='')
          tf.train.write_graph(graph_def, './', pb_path, as_text=False)

pbtxt_to_graphdef(pbtxt_path,pb_path)
```

图 4.35　pbtxt-pb 转换工具

```python
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。读者也可实现完整带BroadCast功能的函数，此为加分项之一
    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 = []
    #通过for循环完成计算，每次循环计算y个数的PowerDifference操作
    for i in range(x_new_shape[0]):
        tmp1 = tmp2 = x[i]-y
        for j in range(input_z-1):
            tmp1 = tmp1*tmp2
        output.append(tmp1)
    output = np.reshape(output,x_shape)
    return output
```

图 4.36　Numpy 算子实现

4.4.5.3　C++ 实现

C++ 实现主要指的是在 TensorFlow 框架中集成用 C++ 编写的算子，以进行高效的模型推理。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 的实现在 tensorflow/core/kernes/cwise_op_power_difference.cc 文件中，其部分代码如图 4.37所示。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 Op 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。

```
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TF自有的BCast Op，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;
        }
      }
    }
};
```

图 4.37　PowerDifference 算子 CPU 实现示例代码

2. **算子注册**：首先在 tensorflow/core/ops 目录下找到对应的 Op 注册文件，TensorFlow 对于 Op 注册位置没有特定的限制，为了尽可能和算子的用途保存一致，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 4.38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 4.39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 4.38　TensorFlow 注册 (1)

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

图 4.39　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（在 PowerDifference 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 4.40所示。该步骤完成后可以采用如图 4.41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 4.40　修改 BUILD 文件

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”

```
#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.执行编译脚本
cd /path/to/tensorflow/source
build_tensorflow-v1.10_cpu.sh
#3.激活虚拟环境
source env.sh
source virtualenv_cpu
```

图 4.41　TensorFlow 的编译

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。示例测试程序如图 4.42所示。

```
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:
      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") #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()
```

图 4.42　单算子比较测试

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理测试。

1. 基于 NumPy 算子进行模型推理

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 计算节点的信息如图 4.43所示。

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

图 4.43 pbtxt 添加 numpy 计算节点

得到新模型后使用 NumPy 完成缺失节点的计算，NumPy 实现的程序如图 4.44所示。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++ 算子进行模型推理

采用 C++ 集成算子 CPU 推理代码如图 4.45所示。由于 PowerDifference 集成到框架内部，对于模型的推理仅需调用一次 Session.run() 即可完成。

4.4.6 实验评估

本次实验中主要考虑框架集成的过程和功能本身的实现，所设定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：在此前的基础上，基于两种算子实现方式进行模型推理的精度正常，且 C++ 实现方式性能要优于 Numpy 实现方式。

本部分得分占本章总得分的 40%。

```python
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())

            # 根据输入名称获得输入的tensor
            input_tensor = sess.graph.get_tensor_by_name('X_content:0')
            # 获取两个输出节点，作为numpy算子的输入
            out_tmp_tensor_1 = sess.graph.get_tensor_by_name('Conv2D_13:0')
            out_tmp_tensor_2 = sess.graph.get_tensor_by_name('moments_15/StopGradient:0'
    )
            #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 =sess.run([out_tmp_tensor_1,out_tmp_tensor_2], feed_dict={
    input_tensor:[X]})
            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)

            # 根据添加的输入节点名称获得输入tensor
            input_tensor_new = sess.graph.get_tensor_by_name('moments_15/PowerDifference
    :0')
            # 完整推理最终输出的tensor
            output_tensor = sess.graph.get_tensor_by_name('add_37:0')
            #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 =sess.run(output_tensor, feed_dict={input_tensor_new:output,input_tensor
    :[X]})
            #结果保存
            img1 = tf.reshape(ret,[256,256,3])
            img_numpy = img1.eval(session=sess)
            cv.imwrite('result_new.jpg',img_numpy)
```

图 4.44　基于 NumPy 算子实现的推理

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（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？

```python
import tensorflow as tf
from tensorflow.python.platform import gfile
import argparse
import numpy as np
import cv2 as cv
import time

def parse_arg():
    parser = argparse.ArgumentParser()
    parser.add_argument('pb', default='style_transfer.pb')
    args = parser.parse_args()
    return args

def run_pb():
    args = parse_arg()
    config = tf.ConfigProto()
    #加载模型
    with tf.gfile.FastGFile(args.pb,'rb') as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())
        tf.import_graph_def(graph_def, name='')
    #读取输入图片
    img = cv.imread("./images/sample.jpg")
    X = cv.resize(img, (256, 256))
    with tf.Session(config=config) as sess:
        #获取计算图，初始化
        sess.graph.as_default()
        sess.run(tf.global_variables_initializer())

        #获取输入输出Tensor
        input_tensor = sess.graph.get_tensor_by_name('X_content:0')
        output_tensor = sess.graph.get_tensor_by_name('add_37:0')

        #run session
        ret =sess.run(output_tensor, feed_dict={input_tensor:[X]})

        #保存结果
        img1 = tf.reshape(ret,[256,256,3])
        img_numpy = img1.eval(session=sess)
        cv.imwrite('result.jpg',img_numpy)
if __name__ == '__main__':
    run_pb()
```

图 4.45　C++ 算子 CPU pb 模型推理实现