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。

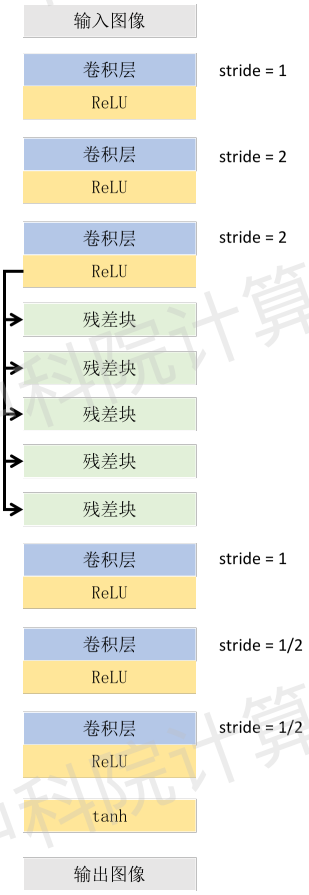

图 4.6　图像转换网络的网络结构

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。

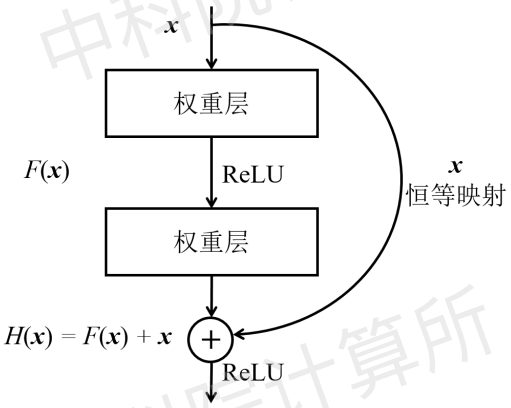

图 4.7　残差块结构

• **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。

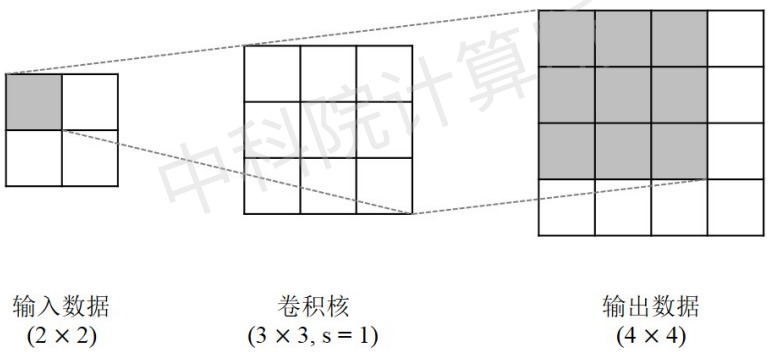

图 4.8　转置卷积

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

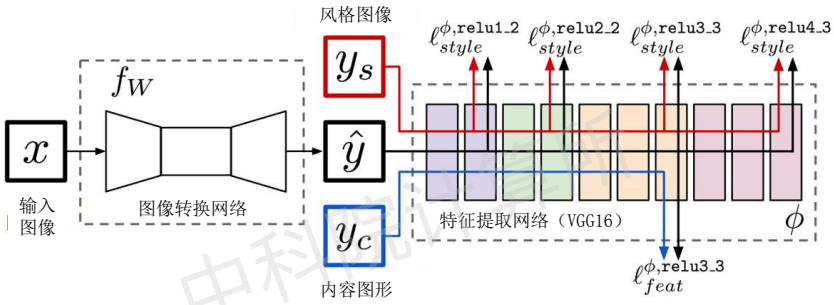

图 4.14　实时图像风格迁移算法的流程[15]

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4.19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

参考文献

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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