第 3 章　深度学习应用实验

《智能计算系统》课程教材中采用图像风格迁移作为驱动范例，将深度学习算法、编程框架、深度学习处理器硬件架构贯穿起来。图像风格迁移是深度学习的一个典型应用，有非实时和实时两种实现方式，其中非实时风格迁移算法使用 VGG19 作为核心网络结构。本章首先介绍如何用 Python 语言实现基于 VGG19 的图像分类，然后介绍如何在 DLP 上实现图像分类，最后介绍如何用 Python 语言实现非实时风格迁移算法。本章的三个实验，均是基于第2章的实验框架扩展而来，同时会复用第2章实验中的全连接层、ReLU 激活函数层等基本单元，在此基础上加入新的基本单元如卷积层、最大池化层。完成本章实验，读者就可以点亮智能计算系统知识树（图1.2）中深度学习部分的知识点。

3.1　基于 VGG19 实现图像分类

3.1.1　实验目的

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

1) 加深对深度卷积神经网络中卷积层、最大池化层等基本单元的理解。

2) 利用 Python 语言实现 VGG19 的前向传播计算，加深对 VGG19 网络结构的理解，为后续风格迁移中使用 VGG19 网络进行特征提取奠定基础。

3) 在第2.1节实验的基础上将三层神经网络扩展为 VGG19 网络，加深对神经网络工程实现中基本模块演变的理解，为后续建立更复杂的综合实验（如风格迁移）奠定基础。

实验工作量：约 30 行代码，约需 3 个小时。

3.1.2　背景介绍

3.1.2.1　卷积神经网络中的基本单元

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

卷积层

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

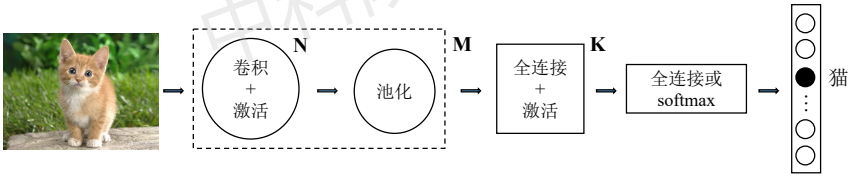

图 3.1　常见卷积神经网络结构

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

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

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

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

$$H_{pad} = H_{in} + 2p \qquad W_{pad} = W_{in} + 2p \tag{3.2}$$

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

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

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

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

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

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

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

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

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

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

最大池化层

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

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

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

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

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

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

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

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

3.1.2.2 VGG19 网络的基本结构

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

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

表 3.1　VGG19 网络基本结构

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

3.1.3　实验环境

硬件环境：CPU。

软件环境：Python 编译环境及相关的扩展库，包括 Python 2.7.12，Pillow 6.0.0，Scipy 0.19.0，NumPy 1.16.0（本实验不需使用 TensorFlow 等深度学习框架）。

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

3.1.4 实验内容

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

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

3.1.5 实验步骤

3.1.5.1 数据加载模块

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

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

图 3.2 VGG19 的数据加载模块实现示例

3.1.5.2 基本单元模块

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

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

卷积层

程序示例如图3.3所示，定义了以下成员函数：

• 层的初始化：需要定义卷积的超参数，包括卷积核的高（或宽）K、输入特征图的通道数 C_{in}、输出特征图通道数 C_{out}、特征图边界扩充大小 p、卷积步长 s 等。

• 参数初始化：卷积层的参数包括权重（卷积核）和偏置。与全连接层类似，通常用高斯随机数初始化权重的值，而将偏置的所有值初始化为 0。

• 前向传播计算：根据公式(3.1)和(3.3)可进行卷积层的前向传播计算。首先利用公式(3.1)对输入特征图进行边界扩充。之后利用公式(3.3)将卷积核与边界扩充后的特征图计算矩阵内积并与偏置相加获得当前位置的输出特征图结果，将卷积核进行滑动获得整个输出特征图的结果。在工程实现中，最简单直接的实现方式是利用四重循环计算输出特征图所有位置的值。由于 VGG19 网络中的所有卷积层都是 3×3 卷积核，即 $K=3$，边界扩充大小 $p=1$，步长 $s=1$，因此 VGG19 网络中的所有卷积层输出特征图的高和宽与输入特征图保持相同。

• 参数加载：从该函数的输入中读取本层的权重 W 和偏置 b。

最大池化层

程序示例如图3.4所示，定义了以下成员函数：

• 层的初始化：需要定义最大池化的超参数，包括池化窗口的高（或宽）K 和池化步长 s。

• 前向传播计算：根据公式(3.7)可计算最大池化层的前向传播结果。输出特征图的某一位置的值为输入特征图的对应池化窗口中的最大值。由于输出特征图的每个位置的值都是输入特征图的对应池化窗口的最大值，因此最简单直接的实现方式是用四重循环来计算输出特征图所有位置的值。

Flatten 层

程序示例如图3.5所示，定义了以下成员函数：

• 层的初始化：flatten 层用于改变特征图的维度，将输入特征图中每个样本的特征平铺成一个向量。初始化 flatten 层时需要定义输入特征图和输出特征图的维度。

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

```python
# file: layer_2.py
class ConvolutionalLayer(object):
    def __init__(self, kernel_size, channel_in, channel_out, padding, stride):
        # 卷积层的初始化
        self.kernel_size = kernel_size
        self.channel_in = channel_in
        self.channel_out = channel_out
        self.padding = padding
        self.stride = stride
    def init_param(self, std=0.01): # 参数初始化
        self.weight = np.random.normal(loc=0.0, scale=std, size=(self.channel_in, self.kernel_size, self.kernel_size, self.channel_out))
        self.bias = np.zeros([self.channel_out])
    def forward(self, input): # 前向传播的计算
        self.input = input # [N, C, H, W]
        height = self.input.shape[2] + self.padding * 2
        width = self.input.shape[3] + self.padding * 2
        self.input_pad = np.zeros([self.input.shape[0], self.input.shape[1], height, width])
        self.input_pad[:, :, self.padding:self.padding+self.input.shape[2], self.padding:self.padding+self.input.shape[3]] = self.input
        height_out = (height - self.kernel_size) / self.stride + 1
        width_out = (width - self.kernel_size) / self.stride + 1
        self.output = np.zeros([self.input.shape[0], self.channel_out, height_out, width_out])
        for idxn in range(self.input.shape[0]):
            for idxc in range(self.channel_out):
                for idxh in range(height_out):
                    for idxw in range(width_out):
                        # TODO: 计算卷积层的前向传播，即特征图与卷积核的内积再加偏置
                        self.output[idxn, idxc, idxh, idxw] = _______________
        return self.output
    def load_param(self, weight, bias): # 参数加载
        self.weight = weight
        self.bias = bias
```

图 3.3　卷积层的实现示例

```python
# file: layer_2.py
class MaxPoolingLayer(object):
    def __init__(self, kernel_size, stride): # 最大池化层的初始化
        self.kernel_size = kernel_size
        self.stride = stride
    def forward(self, input): # 前向传播的计算
        start_time = time.time()
        self.input = input # [N, C, H, W]
        self.max_index = np.zeros(self.input.shape)
        height_out = (self.input.shape[2] - self.kernel_size) / self.stride + 1
        width_out = (self.input.shape[3] - self.kernel_size) / self.stride + 1
        self.output = np.zeros([self.input.shape[0], self.input.shape[1], height_out, width_out])
        for idxn in range(self.input.shape[0]):
            for idxc in range(self.input.shape[1]):
                for idxh in range(height_out):
                    for idxw in range(width_out):
                        # TODO: 计算最大池化层的前向传播，取池化窗口内的最大值
                        self.output[idxn, idxc, idxh, idxw] = _______________
        return self.output
```

图 3.4　最大池化层的实现示例

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

图 3.5　Flatten 层的实现示例

3.1.5.3　网络结构模块

与第2.1节实验类似，本实验的网络结构模块也用一个类来定义 VGG19 神经网络，用类的成员函数来定义 VGG19 的初始化、建立网络结构、神经网络参数初始化等基本操作。VGG19 的网络结构模块的程序示例如图3.6所示，定义了以下成员函数：

• 神经网络初始化：确定神经网络相关的超参数。为方便起见，本实验在网络初始化时仅设定每层的名称，在建立网络结构时再设定每层的具体超参数。

• 建立网络结构：定义整个神经网络的拓扑结构，设定每层的超参数，实例化基本单元模块中定义的层并将这些层堆叠，组成 VGG19 网络结构。根据表3.1中 VGG19 的网络结构和每层的超参数进行实例化。注意每个卷积层和 3 个全连接层中的前 2 个全连接层后面都跟随有 ReLU 层作为激活函数。此外，pool5 层和 fc6 层中间有一个 flatten 层改变特征图的维度。最后是 Softmax 层计算分类概率。

• 神经网络参数初始化：依次调用神经网络中包含参数的层的参数初始化函数。在本实验中，VGG19 中的 16 个卷积层和 3 个全连接层包含参数，因此需要依次调用其参数初始化函数。

3.1.5.4　网络推断模块

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

• 神经网络的前向传播：前向传播的输入是预处理后的图像。首先将预处理后的图像输入到 VGG19 网络的第一层；然后根据之前定义的 VGG19 网络的结构，顺序依次调用每层的前向传播函数，每层的输出作为下一层的输入。由于 VGG19 中的网络层数较多，可以利用网络初始化时定义的层队列，建立循环实现前向传播。

• 神经网络参数的加载：利用官方训练好的 VGG19 模型参数，依次将其中的参数加载到 VGG19 对应的层中。本实验使用的官方模型的下载地址为`http://www.vlfeat.org/matconvnet/models/beta16/imagenet-vgg-verydeep-19.mat`。VGG19 中包含参数的网络层是卷积层和全连接层，可以根据层的编号依次读入对应卷积层和全连接层的权重和偏置。注意在本实验的网络初始化中，在 pool5 层和 fc6 层之间添加了 flatten 层来改变特征图

```python
# file: vgg_cpu.py
class VGG19(object):
    def __init__(self, param_path='imagenet-vgg-verydeep-19.mat'):
        # 神经网络的初始化
        self.param_path = param_path
        self.param_layer_name = (
            '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',
            'flatten', 'fc6', 'relu6', 'fc7', 'relu7', 'fc8', 'Softmax')
    def build_model(self):  # 建立网络结构
        # TODO: 定义 VGG19 的网络结构
        self.layers = {}
        self.layers['conv1_1'] = ConvolutionalLayer(3, 3, 64, 1, 1)
        self.layers['relu1_1'] = ReLULayer()
        self.layers['conv1_2'] = ConvolutionalLayer(3, 64, 64, 1, 1)
        self.layers['relu1_2'] = ReLULayer()
        self.layers['pool1'] = MaxPoolingLayer(2, 2)
        _______________
        self.layers['conv5_4'] = ConvolutionalLayer(3, 512, 512, 1, 1)
        self.layers['relu5_4'] = ReLULayer()
        self.layers['pool5'] = MaxPoolingLayer(2, 2)
        self.layers['flatten'] = FlattenLayer([512, 7, 7], [512*7*7])
        self.layers['fc6'] = FullyConnectedLayer(512*7*7, 4096)
        self.layers['relu6'] = ReLULayer()
        _______________
        self.layers['fc8'] = FullyConnectedLayer(4096, 1000)
        self.layers['Softmax'] = SoftmaxLossLayer()
        self.update_layer_list = []
        for layer_name in self.layers.keys():
            if 'conv' in layer_name or 'fc' in layer_name:
                self.update_layer_list.append(layer_name)
    def init_model(self): # 神经网络参数初始化
        for layer_name in self.update_layer_list:
            self.layers[layer_name].init_param()
```

图 3.6　VGG19 的网络结构模块实现示例

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

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

```
# file: vgg_cpu.py
def load_model(self): # 加载神经网络参数
    params = scipy.io.loadmat(self.param_path)
    self.image_mean = params['normalization'][0][0][0]
    self.image_mean = np.mean(self.image_mean, axis=(0, 1))
    for idx in range(43):
        if 'conv' in self.param_layer_name[idx]:
            weight, bias = params['layers'][0][idx][0][0][0][0]
            # matconvnet: weights dim [height, width, in_channel, out_channel]
            # ours: weights dim [in_channel, height, width, out_channel]
            weight = np.transpose(weight,[2,0,1,3])
            bias = bias.reshape(-1)
            self.layers[self.param_layer_name[idx]].load_param(weight, bias)
        if idx >= 37 and 'fc' in self.param_layer_name[idx]:
            weight, bias = params['layers'][0][idx-1][0][0][0][0]
            weight = weight.reshape([weight.shape[0]*weight.shape[1]*weight.shape[2],
     weight.shape[3]])
            self.layers[self.param_layer_name[idx]].load_param(weight, bias)

def forward(self): # 神经网络的前向传播
    current = self.input_image
    for idx in range(len(self.param_layer_name)):
        current = self.layers[self.param_layer_name[idx]].forward(current)
    return current

def evaluate(self): # 推断函数主体
    prob = self.forward()
    top1 = np.argmax(prob[0])
    print('Classification result: id = %d, prob = %f' % (top1, prob[0, top1]))
```

图 3.7　VGG19 的网络推断模块实现示例

3.1.5.5　实验完整流程

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

```
# file: vgg_cpu.py
if __name__ == '__main__':
    vgg = VGG19()
    vgg.build_model()
    vgg.init_model()
    vgg.load_model()
    vgg.load_image('cat1.jpg')
    vgg.evaluate()
```

图 3.8　VGG19 进行图像分类的完整流程实现示例

3.1.5.6　实验运行

根据第3.1.5.1节 ~ 第3.1.5.5节的描述补全 layer_1.py、layer_2.py、vgg_cpu.py 代码，并通过 Python 运行.py 代码。具体可以参考以下步骤。

1. 环境申请

按照附录B说明申请实验环境并登录云平台，本实验的代码存放在云平台/opt/code_chap_2_3/code_cl目录下。

```
# 登录云平台
ssh root@xxx.xxx.xxx.xxx -p xxxxx
# 进入 code_chap_2_3_student 目录
cd /opt/code_chap_2_3/code_chap_2_3_student
# 初始化环境
source env.sh

```

2. 代码实现

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

```
# 进入实验目录
cd exp_3_1_vgg
# 补全 layers_1.py, layers_2.py, vgg_cpu.py
vim stu_upload/layers_1.py
vim stu_upload/layers_2.py
vim stu_upload/vgg_cpu.py

```

3. 运行实验

```
# 运行完整实验
python main_exp_3_1.py

```

3.1.6　实验评估

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

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

图 3.9　测试猫咪图像示例

本实验的评估标准设定如下：

• 60 分标准：给定卷积层和池化层的前向传播输入矩阵和参数值，可以得到正确的前向传播输出矩阵。

• 80 分标准：建立 VGG19 网络后，给定 VGG19 的网络参数值和输入图像，可以得到正确的 pool5 层输出结果。

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

3.1.7　实验思考

1) 在实现深度神经网络基本单元时，如何确保一个层的实现是正确的?

2) 在实现深度神经网络后，如何确保整个网络的实现是正确的？如果是网络中的某个层计算有误，如何快速定位到有错误的层?

3) 如何计算深度神经网络的每层计算量（乘法数量和加法数量）？如何计算整个网络的前向传播时间和网络中每层的前向传播时间？深度神经网络的每层计算量和每层前向传播时间之间有什么关系?

3.2　基于 DLP 平台实现图像分类

3.2.1　实验目的

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

1) 使用 pycnml 库实现卷积、池化等基本网络模块。

2) 使用提供的 pycnml 库实现 VGG19 网络。

3) 分析并比较 DLP 和 CPU 运行 VGG19 进行图像分类的性能。

实验工作量：约 40 行代码，约需 1 个小时。

3.2.2 实验环境

硬件环境：DLP。

软件环境：pycnml 库、Python 编译环境及相关的扩展库，包括 Python 2.7.12，Pillow 6.0.0，Scipy 0.19.0，NumPy 1.16.0、CNML 高性能算子库、CNRT 运行时库。

数据集：ImageNet。

3.2.3 实验内容

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

3.2.4 实验步骤

3.2.4.1 数据加载模块

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

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

图 3.10 VGG19 的数据加载模块 DLP 实现示例

3.2.4.2 基本单元模块

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

3.2.4.3 网络结构模块

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

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

```
# file: vgg19_demo.py
class VGG19(object):
    def __init__(self):
        # 初始化网络，创建pycnml.CnmlNet() 实例 net
        self.net = pycnml.CnmlNet()
        self.input_quant_params = []
        self.filter_quant_params = []

    def build_model(self,
                    param_path='../data/vgg19_data/imagenet-vgg-verydeep-19.mat',
                    quant_param_path='../data/vgg19_data/vgg19_quant_param_new.npz'):
        self.param_path = param_path
        # 加载量化参数
        params = np.load(quant_param_path)
        input_params = params['input']
        filter_params = params['filter']
        for i in range(0, len(input_params), 2):
            self.input_quant_params.append(pycnml.QuantParam(int(input_params[i]), float(
    input_params[i+1])))
        for i in range(0, len(filter_params), 2):
            self.filter_quant_params.append(pycnml.QuantParam(int(filter_params[i]), float(
    filter_params[i+1])))
        # TODO: 使用 net 的 createXXXLayer 接口搭建 VGG19 网络
        self.net.setInputShape(1, 3, 224, 224)
        # conv1_1
        self.net.createConvLayer('conv1_1', 64, 3, 1, 1, 1, self.input_quant_params[0])
        # relu1_1
        self.net.createReLuLayer('relu1_1')
        # conv1_2
        self.net.createConvLayer('conv1_2', 64, 3, 1, 1, 1, self.input_quant_params[1])
        # relu1_2
        self.net.createReLuLayer('relu1_2')
        # pool1
        ----------------------------
        ----------------------------
        ----------------------------
        # fc8
        self.net.createMlpLayer('fc8', 1000, self.input_quant_params[18])
        # Softmax
        self.net.createSoftmaxLayer('Softmax', 1)
```

3.2.4.4 网络推断模块

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

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

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

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

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

```
# file: vgg19_demo.py
def load_model(self): # 加载神经网络参数
    print('Loading parameters from file ' + self.param_path)
    params = scipy.io.loadmat(self.param_path)
    self.image_mean = params['normalization'][0][0][0]
    self.image_mean = np.mean(self.image_mean, axis=(0, 1))
    count = 0
    for idx in range(self.net.size()):
        if 'conv' in self.net.getLayerName(idx):
            weight, bias = params['layers'][0][idx][0][0][0][0]
            # matconvnet: weights dim [height, width, in_channel, out_channel]
            # ours: weights dim [out_channel, in_channel, height, width]
            weight = np.transpose(weight,[3,2,0,1]).flatten().astype(np.float)
            bias = bias.reshape(-1).astype(np.float)
            self.net.loadParams(idx, weight, bias, self.filter_quant_params[count])
            count += 1
        if 'fc' in self.net.getLayerName(idx):
            weight, bias = params['layers'][0][idx-1][0][0][0][0]
            weight = weight.reshape([weight.shape[0]*weight.shape[1]*weight.shape[2], weight.shape
    [3]])
            weight = np.transpose(weight, [1, 0]).flatten().astype(np.float)
            bias = bias.reshape(-1).astype(np.float)
            self.net.loadParams(idx, weight, bias, self.filter_quant_params[count])
            count += 1

def forward(self): # 神经网络的前向传播
    return self.net.forward()

def get_top5(self, label):
    # 打印推理的时间
    start = time.time()
    self.forward()
    end = time.time()
    print('inference time: %f'%(end - start))
    result = self.net.getOutputData()
    # 打印 top1/5 结果
    top1 = False
    top5 = False
    print('------ Top 5 of ' + self.image + ' ------')
    prob = sorted(list(result), reverse=True)[:6]
    if result.index(prob[0]) == label:
        top1 = True
    for i in range(5):
        top = prob[i]
        idx = result.index(top)
        if idx == label:
            top5 = True
        print('%f - '%top + self.labels[idx].strip())
    return top1,top5

def evaluate(self, file_list): # 推断函数主体
```

```
    top1_num = 0
    top5_num = 0
    total_num = 0
    # 读取标签
    self.labels = []
    with open('synset_words.txt', 'r') as f:
        self.labels = f.readlines()
    # 记录推断所有图片的总时间
    start = time.time()
    with open(file_list, 'r') as f:
        file_list = f.readlines()
        total_num = len(file_list)
        for line in file_list:
            image = line.split()[0].strip()
            label = int(line.split()[1].strip())
            self.load_image(image)
            top1,top5 = self.get_top5(label) # 获取推断结果
            if top1 :
                top1_num += 1
            if top5 :
                top5_num += 1
    end = time.time()
    print('Global accuracy :')
    print('accuracy1: %f (%d/%d) '%(float(top1_num)/float(total_num), top1_num, total_num))
    print('accuracy5: %f (%d/%d) '%(float(top5_num)/float(total_num), top5_num, total_num))
    print('Total execution time: %f'%(end - start))
```

3.2.4.5 实验完整流程

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

```
# file: vgg19_demo.py
if __name__ == '__main__':
    vgg = VGG19()
    vgg.build_model()
    vgg.load_model()
    vgg.evaluate('file_list')
```

图 3.11 VGG19 进行图像分类的完整流程 DLP 实现示例

3.2.4.6 实验运行

根据第3.2.4.1节 ~ 第3.2.4.5节的描述补全 vgg19_demo.py 代码，并通过 Python 运行.py 代码。具体可以参考以下步骤。

1. 环境申请

按照附录B说明申请实验环境并登录云平台，本实验的代码存放在云平台/opt/code_chap_2_3/code_cl 目录下。

```
# 登录云平台
ssh root@xxx.xxx.xxx.xxx -p xxxxx
# 进入 code_chap_2_3_student 目录
cd /opt/code_chap_2_3/code_chap_2_3_student
# 初始化环境
source env.sh
```

2. **代码实现**

补全 stu_upload 中的 vgg19_demo.py 文件。

```
# 进入实验目录
cd exp_3_1_vgg
# 补全 vgg19_demo.py
vim stu_upload/vgg19_demo.py
```

3. **运行实验**

```
# 运行完整实验
python main_exp_3_2.py
```

3.2.5 实验评估

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

本实验的评分标准设定如下：

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

3.2.6 实验思考

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

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

3.3 非实时图像风格迁移

3.3.1 实验目的

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

1) 加深对卷积神经网络的理解，利用 VGG19 模型进行图像特征提取。

2) 使用 Python 语言实现风格迁移中风格和内容损失函数的计算，加深对非实时风格迁移的理解。

3) 使用 Python 语言实现非实时风格迁移中迭代求解风格化图像的完整流程，为后续实现实时风格迁移并建立更复杂的综合实验奠定基础。

实验工作量：约 20 行代码，约需 3 个小时。

3.3.2 背景介绍

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

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

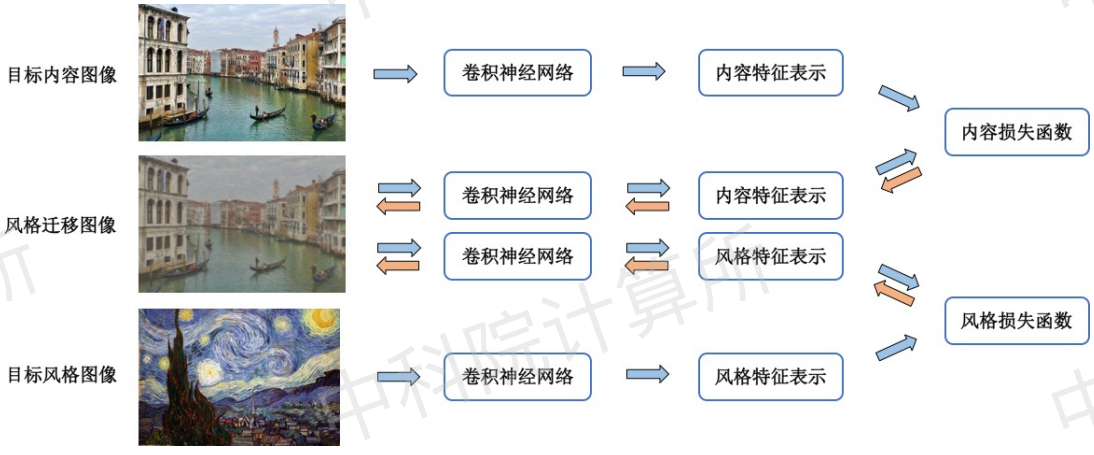

图 3.12　非实时风格迁移

内容损失函数

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

式为

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

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

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

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

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

风格损失函数

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

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

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

$$L^l_{style} = \frac{1}{4NC^2H^2W^2} \sum_{n,i,j} (\boldsymbol{G}^l(n,i,j) - \boldsymbol{A}^l(n,i,j))^2 \tag{3.13}$$

风格损失函数为各层风格损失之和：

$$L_{style} = \sum_l \omega_l L^l_{style} \tag{3.14}$$

其中，ω_l 是计算风格损失时第 l 层损失的权重。

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

迁移图像特征图 $\boldsymbol{X}^l$ 的梯度[①]：

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

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

损失函数

风格迁移的损失函数为内容损失和风格损失的加权和：

$$L_{total} = \alpha L_{content} + \beta L_{style} \tag{3.16}$$

其中，α 和 β 为权重。

Adam 优化器

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

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

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

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

3.3.3 实验环境

硬件环境：CPU。

软件环境：Python 编译环境及相关的扩展库，包括 Python 2.7.12，Pillow 6.0.0，Scipy 0.19.0，NumPy 1.16.0（本实验不需使用 TensorFlow 等深度学习框架）。

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

3.3.4 实验内容

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

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

3.3.5 实验步骤

3.3.5.1 数据加载模块

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

• 数据加载：与第3.1节实验类似，实现从文件中读取图像、缩放图像、图像标准化（减去图像均值）、转换图像矩阵维度等过程。需要注意的是，本实验仅使用 VGG19 模型的卷积层和最大池化层进行特征的提取，VGG19 中的全连接层并没有参与计算，而卷积层和最大池化层的输入特征图的高和宽是可以变化的。因此，可以在数据加载模块中将预处理后图像的分辨率作为超参数，方便灵活调整风格迁移图像的分辨率。

• 生成图像初始化：在目标内容图像中加入随机高斯噪声来初始化风格迁移图像。

• 图像保存：保存最终获得的风格迁移图像。其过程与图像的读取和预处理过程相反，包括转换图像矩阵维度，加上图像均值，缩放图像和保存图像到文件中。

3.3.5.2 基本单元模块

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

卷积层

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

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

```python
# file: exp_3_3_style_transfer.py
def load_image(self, image_dir, image_height, image_width):
    # 数据加载模块
    self.input_image = scipy.misc.imread(image_dir)
    image_shape = self.input_image.shape
    self.input_image = scipy.misc.imresize(self.input_image,[image_height, image_width
    ,3])
    self.input_image = np.array(self.input_image).astype(np.float32)
    self.input_image -= self.image_mean
    self.input_image = np.reshape(self.input_image, [1]+list(self.input_image.shape))
    # input dim [N, channel, height, width]
    self.input_image = np.transpose(self.input_image, [0, 3, 1, 2])
    return self.input_image, image_shape
def get_random_img(content_image, noise):
   # 生成风格迁移初始化图像
    noise_image = np.random.uniform(-20, 20, content_image.shape)
    random_img = noise_image * noise + content_image * (1 - noise)
    return random_img
def save_image(self, input_image, image_shape, image_dir):
    # 保存图像
    input_image = np.transpose(input_image, [0, 2, 3, 1])
    input_image = input_image[0] + self.image_mean
    input_image = np.clip(input_image, 0, 255).astype(np.uint8)
    input_image = scipy.misc.imresize(input_image, image_shape)
    scipy.misc.imsave(image_dir, input_image)
```

图 3.13　非实时风格迁移的数据加载模块实现示例

```python
# file: layer_2.py
def backward(self, top_diff): # 卷积层的反向传播
    self.d_weight = np.zeros(self.weight.shape)
    self.d_bias = np.zeros(self.bias.shape)
    bottom_diff = np.zeros(self.input_pad.shape)
    for idxn in range(top_diff.shape[0]):
        for idxc in range(top_diff.shape[1]):
            for idxh in range(top_diff.shape[2]):
                for idxw in range(top_diff.shape[3]):
                    #TODO: 计算卷积层的反向传播，权重、偏置的梯度和本层损失
                    self.d_weight[:, :, :, idxc] += _____________
                    self.d_bias[idxc] += _____________
                    bottom_diff[idxn, :, idxh*self.stride:idxh*self.stride+self.kernel_size, idxw*
    self.stride:idxw*self.stride+self.kernel_size] += _____________
    bottom_diff = bottom_diff[:, :, self.padding:self.padding+self.input.shape[2], self.padding:
    self.padding+self.input.shape[3]]
    return bottom_diff
```

图 3.14　卷积层反向传播的实现示例

最大池化层

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

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

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

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

图 3.15　池化层反向传播的实现示例

内容损失

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

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

```
# file: layer_3.py
class ContentLossLayer(object):
    def forward(self, input_layer, content_layer): # 前向传播的计算
        #TODO: 计算风格迁移图像和目标内容图像的内容损失
        loss = _______________
        return loss
    def backward(self, input_layer, content_layer): # 反向传播的计算
        #TODO: 计算内容损失的反向传播
        bottom_diff = _______________
        return bottom_diff
```

图 3.16　内容损失计算的实现示例

风格损失

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

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

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

```
# file: layer_3.py
class StyleLossLayer(object):
    def forward(self, input_layer, style_layer): # 前向传播的计算
        # TODO: 计算风格迁移图像和目标风格图像的 Gram 矩阵
        style_layer_reshape = np.reshape(style_layer, [style_layer.shape[0], style_layer.shape[1], -1])
        self.gram_style = _____________
        self.input_layer_reshape = np.reshape(input_layer, [input_layer.shape[0], input_layer.shape[1], -1])
        self.gram_input = np.zeros([input_layer.shape[0], input_layer.shape[1], input_layer.shape[1]])
        for idxn in range(input_layer.shape[0]):
            self.gram_input[idxn, :, :] = _____________
        # TODO: 计算风格迁移图像和目标风格图像的风格损失
        loss = _____________
        return loss
    def backward(self, input_layer, style_layer): # 反向传播的计算
        bottom_diff = np.zeros([input_layer.shape[0], input_layer.shape[1], input_layer.shape[2]*input_layer.shape[3]])
        for idxn in range(input_layer.shape[0]):
            # TODO: 计算风格损失的反向传播
            bottom_diff[idxn, :, :] = _____________
        bottom_diff = np.reshape(bottom_diff, input_layer.shape)
        return bottom_diff
```

图 3.17　风格损失层的实现示例

Adam 优化器

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

Adam 优化器的程序示例如图 3.18所示。

3.3.5.3　网络结构模块

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

```python
# file: exp_3_3_style_transfer.py
class AdamOptimizer(object):
    def __init__(self, lr, diff_shape): # Adam优化器的初始化
        self.beta1 = 0.9
        self.beta2 = 0.999
        self.eps = 1e-8
        self.lr = lr
        self.mt = np.zeros(diff_shape)
        self.vt = np.zeros(diff_shape)
        self.step = 0
    def update(self, input, grad): # 参数更新过程
        self.step += 1
        self.mt = self.beta1 * self.mt + (1 - self.beta1) * grad
        self.vt = self.beta2 * self.vt + (1 - self.beta2) * np.square(grad)
        mt_hat = self.mt / (1 - self.beta1 ** self.step)
        vt_hat = self.vt / (1 - self.beta2 ** self.step)
        # TODO: 利用梯度的一阶矩和二阶矩的无偏估计更新风格迁移图像
        output = _______________
        return output
```

图 3.18　Adam 优化器的实现示例

```python
# file: exp_3_3_style_transfer.py
class VGG19(object):
    def __init__(self, param_path='imagenet-vgg-verydeep-19.mat'):
        # 神经网络的初始化
        self.param_path = param_path
        self.param_layer_name = (
            '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')
    def build_model(self): # 建立网络结构
        # TODO: 建立 VGG19 网络结构
        self.layers = {}
        self.layers['conv1_1'] = ConvolutionalLayer(3, 3, 64, 1, 1)
        self.layers['relu1_1'] = ReLULayer()
        self.layers['conv1_2'] = ConvolutionalLayer(3, 64, 64, 1, 1)
        self.layers['relu1_2'] = ReLULayer()
        self.layers['pool1'] = MaxPoolingLayer(2, 2)

        _______________
        self.layers['conv5_4'] = ConvolutionalLayer(3, 512, 512, 1, 1)
        self.layers['relu5_4'] = ReLULayer()
        self.layers['pool5'] = MaxPoolingLayer(2, 2)
        self.update_layer_list = []
        for layer_name in self.layers.keys():
            if 'conv' in layer_name or 'fc' in layer_name:
                self.update_layer_list.append(layer_name)
    def init_model(self): # 神经网络参数初始化
        for layer_name in self.update_layer_list:
            self.layers[layer_name].init_param()
```

图 3.19　非实时风格迁移中 VGG19 的网络结构模块实现示例

3.3.5.4 网络训练模块

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

• 神经网络的前向传播：通常的神经网络前向传播过程如第2.1节实验中所介绍的，将预处理后的图像输入到神经网络的第一层中，再根据之前定义的网络结构顺序依次调用每层的前向传播函数，然后将每层的输出作为下一层的输入，直到得到最后一层的输出结果。但在非实时风格迁移中，计算内容损失和风格损失函数时，可能会用到中间层的特征图，因此本实验中的前向传播函数会将计算内容/风格损失需要使用的层作为输入参数，利用一个字典记录所有这些层的输出结果。

• 神经网络的反向传播：通常的神经网络反向传播过程是利用神经网络最后一层的输出与标记计算损失，之后利用链式法则逆序逐层计算损失函数对每层输入及参数的偏导（损失及参数梯度），最后得到神经网络所有层的参数梯度，如第2.1节实验。在非实时风格迁移的反向传播过程与通常的神经网络存在两方面的区别：一方面，非实时风格迁移在反向传播时不需要计算每层的参数梯度，仅需计算每层的损失，用最终得到的第一层的损失作为风格迁移图像的梯度对其进行更新。另一方面，计算内容损失函数和风格损失函数时需要对多个中间层的特征图计算损失函数，而不一定只对最后一层特征图计算损失函数。因此在实现反向传播时，首先定位所有计算损失的中间层位置，然后以该层开始逆序计算前面每一层的损失，最终得到第一层的损失。

• 加载神经网络参数：此过程与第3.1节实验中加载 VGG19 官方模型参数的过程基本一致。不同之处在于仅需加载卷积层的参数和预处理图像时使用的图像均值，可以省略全连接层的参数。

• 神经网络训练函数主体：由于非实时风格迁移是对输入的风格迁移图像进行更新，而不是对神经网络参数进行更新。为方便实现，将训练函数主体放在下一小节 “实验完整流程” 中。

3.3.5.5 实验完整流程

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

• 首先确定超参数，包括计算内容损失和风格损失函数时使用 VGG19 的哪些层、数据预加载模块相关的图像缩放分辨率、训练有关的学习率大小、迭代次数、损失函数权重系数等。在本实验中，计算内容损失函数使用的内容损失层为 relu4_2 层，计算风格损失函数使用的风格损失层为 relu1_1、relu2_1、relu3_1、relu4_1、relu5_1 层。

• 其次建立 VGG19 的网络结构并从官方模型中加载参数，同时实例化非实时风格迁移中的内容损失计算、风格损失计算和 Adam 优化器。之后读取给定的内容图像和风格图像

```python
# file: exp_3_3_style_transfer.py
def forward(self, input_image, layer_list): # 前向传播的计算
    current = input_image
    layer_forward = {}
    for idx in range(len(self.param_layer_name)):
        # TODO: 计算 VGG19 网络的前向传播
        current = _____________
        if self.param_layer_name[idx] in layer_list:
            layer_forward[self.param_layer_name[idx]] = current
    return layer_forward

def backward(self, dloss, layer_name): # 反向传播的计算
    layer_idx = list.index(self.param_layer_name, layer_name)
    for idx in range(layer_idx, -1, -1):
        # TODO: 计算 VGG19 网络的反向传播
        dloss = _____________
    return dloss
```

图 3.20　非实时风格迁移的网络训练模块实现示例

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

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

3.3.5.6　实验运行

根据第3.3.5.1节～第3.3.5.5节的描述补全 layer_1.py、layer_2.py、layer_3.py、style_transfer.py 代码，并通过 Python 运行.py 代码。具体可以参考以下步骤。

1. 环境申请

按照附录B说明申请实验环境并登录云平台，本实验的代码存放在云平台/opt/code_chap_2_3/code_cl目录下。

```
# 登录云平台
ssh root@xxx.xxx.xxx.xxx -p xxxxx
# 进入 code_chap_2_3_student 目录
cd /opt/code_chap_2_3/code_chap_2_3_student
# 初始化环境
source env.sh

```

2. 代码实现

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

```
# 进入实验目录
cd exp_3_1_vgg
# 补全 layer_1.py, layer_2.py, layer_3.py, style_transfer.py
vim stu_upload/layer_1.py
```

```python
# file: exp_3_3_style_transfer.py
if __name__ == '__main__':
    CONTENT_LOSS_LAYERS = ['relu4_2']
    STYLE_LOSS_LAYERS = ['relu1_1', 'relu2_1', 'relu3_1', 'relu4_1', 'relu5_1']
    NOISE = 0.5
    ALPHA, BETA = 1, 500
    TRAINTEP = 2001
    LEARNING_RATE = 1.0
    IMAGE_HEIGHT, IMAGE_WIDTH = 192, 320

    vgg = VGG19()
    vgg.build_model()
    vgg.init_model()
    vgg.load_model()
    content_loss_layer = ContentLossLayer()
    style_loss_layer = StyleLossLayer()
    adam_optimizer = AdamOptimizer(LEARNING_RATE, transfer_image.shape)

    content_image, content_shape = vgg.load_image('content.jpg', IMAGE_HEIGHT,
     IMAGE_WIDTH)
    style_image, _ = vgg.load_image('style.jpg', IMAGE_HEIGHT, IMAGE_WIDTH)
    content_layers = vgg.forward(content_image, CONTENT_LOSS_LAYERS)
    style_layers = vgg.forward(style_image, STYLE_LOSS_LAYERS)
    transfer_image = get_random_img(content_image, NOISE)

    for step in range(TRAINTEP):
        transfer_layers = vgg.forward(transfer_image, CONTENT_LOSS_LAYERS +
     STYLE_LOSS_LAYERS)
        content_loss = np.array([])
        style_loss = np.array([])
        content_diff = np.zeros(transfer_image.shape)
        style_diff = np.zeros(transfer_image.shape)
        for layer in CONTENT_LOSS_LAYERS:
            # TODO: 计算内容损失的前向传播
            current_loss = _____________
            content_loss = np.append(content_loss, current_loss)
            # TODO: 计算内容损失的反向传播
            dloss = content_loss_layer.backward(transfer_layers[layer], content_layers[
     layer])
            content_diff += _____________
        for layer in STYLE_LOSS_LAYERS:
            # TODO: 计算风格损失的前向传播
            current_loss = _____________
            style_loss = np.append(style_loss, current_loss)
            # TODO: 计算风格损失的反向传播
            dloss = style_loss_layer.backward(transfer_layers[layer], style_layers[layer
     ])
            style_diff += _____________
        total_loss = ALPHA * np.mean(content_loss) + BETA * np.mean(style_loss)
        image_diff = ALPHA * content_diff / len(CONTENT_LOSS_LAYERS) + BETA * style_diff
     / len(STYLE_LOSS_LAYERS)
        # TODO: 利用 Adam 优化器对风格迁移图像进行更新
        transfer_image = _____________
        if step % 20 == 0:
            print('Step %d, loss = %f' % (step, total_loss), content_loss, style_loss)
            vgg.save_image(transfer_image, content_shape, 'output/output_' + str(step) +
     '.jpg')
```

图 3.21　非实时风格迁移的完整流程实现示例

```
vim stu_upload/layer_2.py
vim stu_upload/layer_3.py
vim stu_upload/style_transfer.py
```

3. 运行实验

```
# 运行完整实验
python main_exp_3_3.py
```

3.3.6 实验评估

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

图 3.22　目标风格图像示例

图 3.23　目标内容图像示例

本实验的评估标准设定如下：

• 60 分标准：正确实现利用四重循环计算卷积层和池化层的前向传播和反向传播过程。给定卷积层和最大池化层的前向传播和反向传播输入矩阵和参数值，可以得到正确的前向传播输出矩阵和反向传播输出梯度。同时分别给出卷积层和最大池化层正确的前向传播和反向传播时间。

• 80 分标准：正确实现内容损失函数和风格损失函数的计算，计算得出风格迁移后的图片。给定生成图像、目标内容图像和目标风格图像，可以计算得到正确的内容损失值和

图 3.24　初始化风格迁移图像示例

图 3.25　迭代 20 次后的风格迁移图像示例

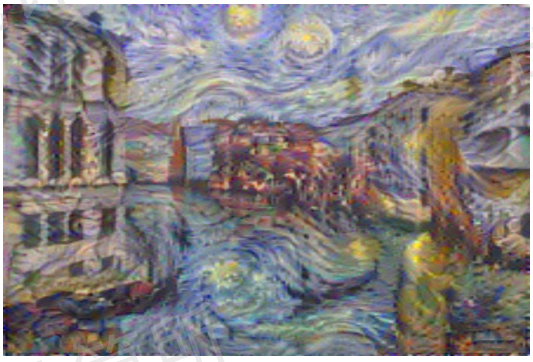

图 3.26　迭代 1000 次后的风格迁移图像示例

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

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

3.3.7　实验思考

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

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

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

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

3.3.8 延伸拓展

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

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

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

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

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

图 3.27　卷积层的并行化实现示例

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.