基于 Tensorflow2.X 的“图像风格迁移”实例学习

一、TensorFlow 简介

TensorFlow 是 Google Brain 基于 DistBelief 进行研发的第二代人工智能学习系统，相对于深度学习基础架构 DistBelief 进行了各方面的改进，不仅在性能上有显著改进，构架灵活性和可移植性也得到增强。

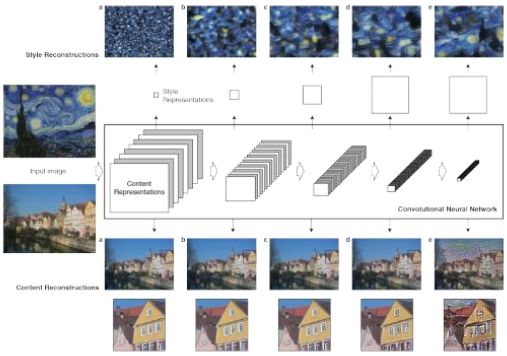
Tensor 的意思是张量，换言之就是 N 维数组，Flow 的意思是流，TensorFlow 就是一种基于数据流图的计算，是张量从流图的一端流动到另一端的计算过程，这也是 TensorFlow 名字的由来。

TensorFlow 1.X 版本一直采用的是 Graph 模式，即先构建一个计算图，然后需要开启 Session，喂进实际的数据才真正执行得到结果。而在 2.X 版本的更新中，graph，session，run，placeholder，feed_dict 这些与静态模型构建息息相关的函数在 2.X 版本中不再使用了。2.X 版本中 TensorFlow 使用了和原生 Python 一致的 Eager 执行，使计算更简洁。

二、图像风格迁移简介

所谓图像风格迁移，是指利用算法学习著名画作的风格，然后再把这种风格应用到另外一张图片上的技术。本实例采用了 VGG19 模型，它是一个用于图像分类的网络，在这个网络中，前面的卷积层是为了提取特征，后面的全连接层把所得特征转换为类别概率，从而输出图像类别。而图像风格迁移正好与之相反，输入特征，输出对应这种特征的图片。具体来说，风格迁移是使用卷积层的中间特征还原出对应这种特征的原始图像的一种技术。

2-1 网络上风格迁移的实例图像



2-2 风格迁移训练结构

三、代码讲解

3.1 图片处理模块

一张 RGB 三通道的彩色图像可以看成一个经过了压缩编码的三维矩阵，矩阵中的数字代表图像的像素值。要获取图像的信息，首先要对图像进行解码，将图像还原成可计算的三维矩阵，然后经过一系列预处理后供模型训练使用，最后再对图像进行编码，输出可视化图像。

在样本预处理时通常要进行数据归一化，本实例采用的是 ImageNet 数据集，其中图像集每个通道的均值 mean 和方差 std 序列分别是[0.485,0.456,0.406]、[0.229,0.224,0.225]。图像归一化：（样本 - 均值） / 方差。

```
###func of img process start###
image_mean = tf.constant([0.485, 0.456, 0.406])      #经典网络img平均值
image_std = tf.constant([0.299, 0.224, 0.225])       #经典网络img标准差


#图片归一化
def normalization(image):
    return (image - image_mean) / image_std


#加载并处理图片
def load_image(image_path, width=WIDTH, height=HEIGHT):
    img = tf.io.read_file(image_path)                    #加载文件
    img = tf.image.decode_jpeg(img, channels=3)          #解码图片，彩色图片cannel=3
    img = tf.image.resize(img, [height, width])          #修改图片大小
    img = img / 255.
    img = normalization(img)                             #归一化
    img = tf.reshape(img, [1, height, width, 3])
    return img


#保存图片
def save_image(image, filename):
    img = tf.reshape(image, image.shape[1:])#height,width,channels
    img = img * image_std + image_mean      #反归一化
    img = img * 255.                        #修改图片大小
    img = tf.cast(img, tf.int32)            #数据类型转换，32位整型
    img = tf.clip_by_value(img, 0, 255)     #将张量数值限定在0-255即rgb
    img = tf.cast(img, tf.uint8)            #数据类型转换，8位无符号整型
    img = tf.image.encode_jpeg(img)         #编码图片，彩色图片cannel=3
    tf.io.write_file(filename, img)         #写入图片文件
### func of img process end ###
```

3.1-1 图片处理模块代码及注释

函数介绍：

（1）图像文件读写

tf.io.read_file(filename)：读取并输出输入文件名的全部内容。

tf.io.write_file(filename)：将内容写入输入文件名的文件中。

（2）图像编码解码

tf.image.decode_jpeg(image):JPEG(JPG)解码图像。

tf.image.encode_jpeg(image):JPEG(JPG)编码图像。

（3）图像尺寸调整

tf.image.resize (image, size, method)：使用指定的 method 调整 images 为 size。

（4）tf.clip_by_value(A, min, max)：输入一个张量 A，把 A 中的每一个元素的值都压缩在 min 和 max 之间。小于 min 的让它等于 min，大于 max 的元素的值等于 max

（5）tf.cast()函数的作用是执行 tensorflow 中张量数据类型转换，比如读入的图片如果是 int8 类型的，一般在要在训练前把图像的数据格式转换为 float32。

3.2 模型初始化模块

本实例采用的是 VGG19 模型，实例直接加载 ImageNet 上预训练的 VGG19 模型，提取需要被用到的 VGG 层，并且用这些层创建新的模型。

```
#创建并初始化vgg19模型
def get_vgg19_model(layers):
    vgg = tf.keras.applications.VGG19(include_top=False, weights='imagenet')  #加载imagenet上预训练的vgg19
    outputs = [vgg.get_layer(layer).output for layer in layers]  #提取需要被用到的vgg的层到outputs
    model = tf.keras.Model([vgg.input, ], outputs)               #使用outputs创建新的模型
    model.trainable = False                                      #锁死参数，不进行训练
    return model
```

3.2-1 模型初始化代码

3.3 损失计算模块

3.3.1 图像内容损失

内容损失被定义为

$$Loss_{content} = \frac{\sum_{i,j}(F_{ij} - P_{ij})^2}{2MN}$$

其中，F 表示卷积层的输出特征，P 表示原始图像的特征。

```
#计算指定层上两个特征之间的内容loss
def _compute_content_loss(noise_features, target_features):
    content_loss = tf.reduce_sum(tf.square(noise_features - target_features))
    x = 2. * M * N              #计算系数
    return content_loss / x


#计算并减小当前图片的内容loss
def compute_content_loss(noise_content_features):
    content_losses = []         #初始化内容损失
    #加权计算内容损失
    for (noise_feature, factor), (target_feature, _) in zip(noise_content_features, target_content_features):
        layer_content_loss = _compute_content_loss(noise_feature, target_feature)
        content_losses.append(layer_content_loss * factor)
    return tf.reduce_sum(content_losses)
```

3.3.1-1 内容损失计算

函数介绍：

（1）tf.square(tensor):对每个元素求平方。

（2）tf.reduce_sum(tensor)：用于计算张量 tensor 沿着某一维度的和，可以在求和后降维。

（3）zip(iterable) ：从参数中的多个迭代器取元素组合成一个新的迭代器。

3.3.2 图像风格损失

事实证明，图像的风格可以通过不同特征图上的平均值和相关性来描述，为了反映度量各个维度的特征以及与其他维度之间的关系，风格迁移算法采用了使用图像卷积层特征的 Gram 矩阵来表示图像风格。Gram 矩阵是关于一组向量的内积的对称矩阵，可以反映出该组向量中各个向量之间的某种关系。Gram 矩阵可由$G_{ij} = \sum_k F_{ik} F_{jk}$得出，即$G_{ij} = (F_i)^T F_j$。其中 F 表示卷积层的输出，G 表示经计算得到的 Gram 矩阵。

```
def gram_matrix(feature):
    x = tf.transpose(feature, perm=[2, 0, 1])
    x = tf.reshape(x, (x.shape[0], -1))
    return x @ tf.transpose(x)
```

3.3.2-1 Gram 矩阵计算

函数介绍：

（1）tf.transpose(tensor,perm)：数组转置函数，参数 perm 控制转置操作，如图中 perm=[2,0,1]是将 channel 维度提到最前。

（2）tf.reshape(tensor,shape)：将 tensor 变换为参数 shape 形式。

风格损失被定义为

$$Loss_{style} = \frac{\sum_{i,j}(A_{ij} - G_{ij})^2}{4N^2M^2}$$

其中，A 表示原始图像某一层卷积的 Gram 矩阵，G 表示风格图像对应卷积层的 Gram 矩阵。分母上的$4M^2N^2$是一个归一化项，目的是防止风格损失的数量级相比内容损失过大。

```
#计算指定层上两个特征之间的风格loss
def _compute_style_loss(noise_feature, target_feature):
    noise_gram_matrix = gram_matrix(noise_feature)
    style_gram_matrix = gram_matrix(target_feature)
    style_loss = tf.reduce_sum(tf.square(noise_gram_matrix - style_gram_matrix))
    x = 4. * (M ** 2) * (N ** 2)                #计算系数
    return style_loss / x


#计算并返回图片的风格loss
def compute_style_loss(noise_style_features):
    style_losses = []
    for (noise_feature, factor), (target_feature, _) in zip(noise_style_features, target_style_features):
        layer_style_loss = _compute_style_loss(noise_feature, target_feature)
        style_losses.append(layer_style_loss * factor)
    return tf.reduce_sum(style_losses)
```

3.3.2-2 风格损失计算

3.3.3 总损失

定义总的损失函数为：

$$Loss_{total} = \alpha Loss_{content} + \beta Loss_{style}$$

其中α、β是平衡两个损失的超参数，如果α偏大，还原的图像会更接近原始图像。使用总的损失函数可以组合原始内容图像和原始风格图像，这实现了图像风格的迁移。本实例中使用的α=1， β=100。

```
def total_loss(noise_features):
    content_loss = compute_content_loss(noise_features['content'])
    style_loss = compute_style_loss(noise_features['style'])
    return content_loss * CONTENT_LOSS_FACTOR + style_loss * STYLE_LOSS_FACTOR
```

3.3.3-1 总损失计算

3.4 训练模块

3.4.1 训练函数

根据内容图像随机生成噪声图像作为输入图像，每进行一次训练利用Adam优化器进行一次梯度下降更新噪声图像。

```
optimizer = tf.keras.optimizers.Adam(LEARNING_RATE)     #使用Adma优化器
#基于内容图片随机生成一张噪声图片
noise_image = tf.Variable((content_image + np.random.uniform(-0.2, 0.2, (1, HEIGHT, WIDTH, 3))) / 2)

#一次迭代过程
@tf.function        #使用tf.function加速训练
def train_one_step():
    #求loss
    with tf.GradientTape() as tape:
        noise_outputs = model(noise_image)
        loss = total_loss(noise_outputs)
    grad = tape.gradient(loss, noise_image)              #求梯度
    optimizer.apply_gradients([(grad, noise_image)])     #梯度下降，更新噪声图片
    return loss
```

3.4.1-1 训练函数代码

函数介绍：

（1）tf.GradientTape():根据某个函数的输入变量来计算它的导数，并记录在一个磁带 tape 上。

（2）tf.GradientTape.gradient(target,source)：根据 tape 上面的上下文来计算某个或者某些 tensor 的梯度。

（3）apply_gradients(grads_and_vars)：把计算出来的梯度更新到变量上面去。

3.4.2 可视化显示

为更好地展示训练过程，实例采用了 tqdm 库提示训练进度，其中 total 代表预期迭代次数，即每个 epoch 的训练次数，desc 代表进度条标题。

```
for epoch in range(EPOCHS):
    with tqdm(total=STEPS_PER_EPOCH, desc='Epoch {}/{}'.format(epoch + 1, EPOCHS)) as pbar:
        for step in range(STEPS_PER_EPOCH):
            _loss = train_one_step()
            pbar.set_postfix({'loss': '%.4f' % float(_loss)})
            pbar.update(1)
        save_image(noise_image, '{}/{}.jpg'.format(OUTPUT_DIR, epoch + 1))
```

3.4.2-1 训练代码

```
Epoch 11/20: 100%|██████████| 100/100 [01:58<00:00,  1.32s/it, loss=4044.7056]
Epoch 12/20: 100%|██████████| 100/100 [02:01<00:00,  1.20s/it, loss=3544.0085]
Epoch 13/20: 100%|██████████| 100/100 [02:03<00:00,  1.23s/it, loss=3367.3074]
Epoch 14/20: 100%|██████████| 100/100 [02:00<00:00,  1.19s/it, loss=4137.4409]
Epoch 15/20:  70%|███████   | 70/100 [01:24<00:36,  1.21s/it, loss=2929.0583]
```

3.4.2-2 可视化进度条

函数介绍：

（1）进度条信息设置

pbar.set_postfix(content)：设置进度条右边显示的信息。

（2）进度条长度更新

pbar.update(length)：每次更新进度条的长度。

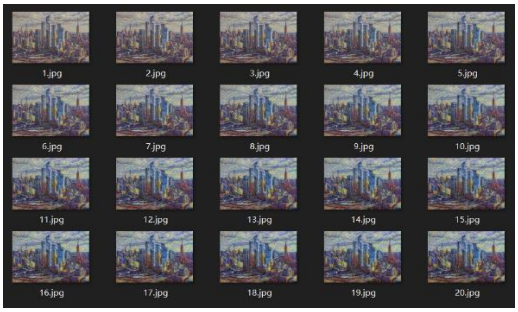
四、结果展示

运用以上代码，在 data 文件夹中导入内容图像 content.jpg，风格图像 style.jpg。运行主程序，将在 output 文件夹中生成 20 个 epoch（每个 epoch 训练 100 次）的风格迁移结果。这里以梵高的著名画作《星空》作为风格图像，分别用城市和北理校园中拍摄的一张照片作为内容图像，进行风格迁移的训练。

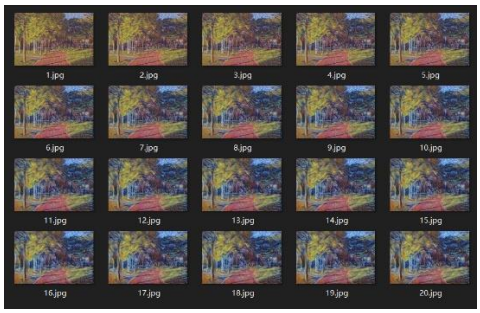
4-1 风格图像《星空》

4-2 内容图像《城市》

4-3 内容图像《北理一角》



4-4 城市+星空训练结果



4-5 北理一角+星空训练结果

可以观察到，在第一个 epoch 时，图像的损失较大，风格图像并不明显，在随着训练次数的增加，损失不断减少，生成的图像将越来越接近想要绘制的风格。

4-6 星空风格的图像展示