作者：Charles

本文档主要参考相关网络资源，并对其进行整理修正获得。如有疑问请联系 qq:1159254961。

文档完全免费，供相关人员参考学习与交流。

文章主要参考自：

http://pytorch.org/tutorials/advanced/neural_style_tutorial.html#

作者：Alexis Jacq

Neural Transfer

一.这是什么？

神经风格或者神经转移是一种将输入的内容图像转换为使用输入的风格图像的艺术风格的内容图像的算法（左边的为内容图像，中间的为风格图像，右边的为转换后的图像）：

二.工作原理

原理很简单：我们定义了两个距离，一个代表图像内容（D_C），一个代表图像风格（D_S）。D_C衡量两幅图像之间的内容差异程度，D_S衡量两幅图像之间的风格差异程度。然后，我们取第三幅图像作为输入（比如含噪音的空白图），我们对其进行转换，使得它的内容与内容图像的距离最小，同时它的风格与风格图像的距离最小。

具体实现：

进一步的理解需要涉及一些数学相关的内容。C_{nn}为一个预先训练好的深度卷积神经网络，X为任意一幅图像。C_{nn}(X)的输入为X（含所有层的特征图）。令$F_{XL} \in C_{nn}$(X)为第L层的特征图，将其全部矢量化并拼接为一个向量。我们简单地定义X的内容在L层为F_{XL}。之后，如果Y是另一张与X相同大小的图像，我们定义两幅图在第L层的内容距离为：

$$D_C^L(X,Y) = ||F_{XL} - F_{YL}||^2 = \sum_i (F_{XL}(i) - F_{YL}(i))^2$$

$F_{XL}(i)$为F_{XL}的第i个元素。风格的定义相对比较困难。令F_{XL}^k(k ≤ K)为第L层的K个特征图的第k个向量。X在第L层的风格G_{XL}被定义为所有特征图向量F_{XL}^k(k ≤ K)的 Gram 积。换句话说，G_{XL}是一个K × K的矩阵，$G_{XL}(k,l)$在G_{XL}的第k行第l列，是F_{XL}^k和F_{XL}^l的矢量积：

$$G_{XL}(k,l) =< F_{XL}^k, F_{XL}^l >= \sum_i F_{XL}^k(i).F_{XL}^l(i)$$

其中$F_{XL}^k(i)$为F_{XL}^k的第i个元素。我们可以发现$G_{XL}(k,l)$为特征图k和l的相关性度量。也就是说，G_{XL}代表X在第L层的特征图的相关矩阵。注意G_{XL}的大小只取决于特征图的数量，而不是X的大小。如果Y是任意尺寸的其他图像，我们定义L层的风格距离为：

$$D_S^L(X,Y) = ||G_{XL} - G_{YL}||^2 = \sum_{k,l} (G_{XL}(k,l) - G_{YL}(k,l))^2$$

现在我们的目标是最小化需要变化的图像X和目标内容图像C之间的$D_C(X,C)$以及X和目标风格图像S之间的$D_S(X,S)$，它们都在几个层中计算，我们在每个需要的层中计算并求和的每个距离的梯度（相对于X的导数）：

$$\nabla_{extittotal}(X,S,C) = \sum_{L_C} w_{CL_C}.\nabla_{extitcontent}^{L_C}(X,C) + \sum_{L_S} w_{SL_S}.\nabla_{extitstyle}^{L_S}(X,S)$$

其中L_C和L_S是各自风格和内容需要的层，w_{CL_C}和w_{SL_S}为每个需要的层的和内容或风格相关的权重。接着我们定义一个X的变化：

$$X \leftarrow X - \alpha\nabla_{extittotal}(X,S,C)$$

更多详细的内容可以参考提供的论文。

三. 具体实现

（1）相关包

- torch，torch.nn，numpy：使用 PyTorch 必不可少的包；
- torch.autograd.Variable：对变量梯度的动态计算；
- torch.optim：有效的梯度下降；
- PIL，PIL.Image，matplotlib.pyplot：导入和显示图片；
- torchvision.transforms：处理 PIL 图像，将其转为 torch 张量；
- torch.models：训练或导入预训练模型；
- copy：深度复制模型或系统包。

```
from __future__ import print_function

import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.optim as optim

from PIL import Image
import matplotlib.pyplot as plt

import torchvision.transforms as transforms
import torchvision.models as models

import copy
```

（2）Cuda

如果你电脑上有 GPU，则最好在 GPU 上运行该算法，尤其当网络比较大时（比如 VGG）。为此，我们使用 torch.cuda.is_available()检测电脑上是否有可用 GPU。然后，我们利用.cuda()方法将运算移至 GPU。当我们想移回 CPU 时（例如使用 numpy 时），我们可以使用.cuda()方法。最后.type(dtype)将 torch.FloatTensor 转换为 torch.cuda.FloatTensor。

```
use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

（3）载入图像

为了简化应用，我们导入相同维度的风格和内容图像。之后，我们将它们缩放为需要的输出的图像尺寸（例如 128 或者 512），将它们转为 torch 张量，准备送入神经网络。

```
# desired size of the output image
imsize = 512 if use_cuda else 128  # use small size if no gpu

loader = transforms.Compose([
    transforms.Scale(imsize),  # scale imported image
    transforms.ToTensor()])  # transform it into a torch tensor


def image_loader(image_name):
    image = Image.open(image_name)
    image = Variable(loader(image))
    # fake batch dimension required to fit network's input dimensions
    image = image.unsqueeze(0)
    return image


style_img = image_loader("images/picasso.jpg").type(dtype)
content_img = image_loader("images/dancing.jpg").type(dtype)

assert style_img.size() == content_img.size(), \
    "we need to import style and content images of the same size"
```

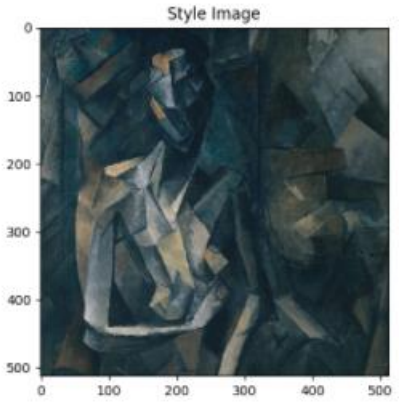
输入的 PIL 图像像素值在 0 到 255 之间，将其转为 torch 张量后，它们的值变为 0 到 1 之间。其中一个重要的细节是：从 torch 库导入的神经网络是通过 0-1 之间取值的张量图像训练的。如果你尝试用 0-255 之间取值的张量图像导入网络，获得的特征图将毫无意义。Caffe 库中预训练的网络则并非如此，它们使用 0-255 之间取值的张量训练。

（4）图像显示

我们使用 plt.imshow 函数显示图像。我们需要先将图像转为 PIL 图像：

```
unloader = transforms.ToPILImage()  # reconvert into PIL image

plt.ion()

def imshow(tensor, title=None):
    image = tensor.clone().cpu()  # we clone the tensor to not do changes on it
    image = image.view(3, imsize, imsize)  # remove the fake batch dimension
    image = unloader(image)
    plt.imshow(image)
    if title is not None:
        plt.title(title)
    plt.pause(0.001) # pause a bit so that plots are updated


plt.figure()
imshow(style_img.data, title='Style Image')

plt.figure()
imshow(content_img.data, title='Content Image')
```

结果：



（5）内容损失

内容损失函数将X输入网络后在L层处的特征图F_{XL}作为输入，返回该图像和内容图像之间的加权内容距离$w_{CL}.D_C^L(X,C)$。因此，权重w_{CL}和目标内容F_{CL}为函数的参数。我们将这个函数作为一个以这些参数作为输入的 torch 模块。距离$||F_{XL}-F_{YL}||^2$为两组特征图之间的均方误差，可以用 nn.MSELoss 计算。

我们在每个需要的层添加内容损失作为神经网络的附加模块。这样我们可以随时输入网络一个图像X，所有的内容损失将在所需要的层中计算，同时由于自动求导机制，所有的梯度都会被计算出来。因此，我们只需要把我们模型的返回值作为 forward 方法的输入：该模块成为神经网络的“透明层”。计算出的损失被保存为模块的参数。

最后，我们定义一个虚假的 backward 方法，它只是调用 nn.MSELoss 的反向方法来重构梯度。

此方法返回计算出的损失：当展示风格和内容损失的梯度下降过程时，这将非常有用。

```
class ContentLoss(nn.Module):

    def __init__(self, target, weight):
        super(ContentLoss, self).__init__()
        # we 'detach' the target content from the tree used
        self.target = target.detach() * weight
        # to dynamically compute the gradient: this is a stated value,
        # not a variable. Otherwise the forward method of the criterion
        # will throw an error.
        self.weight = weight
        self.criterion = nn.MSELoss()

    def forward(self, input):
        self.loss = self.criterion(input * self.weight, self.target)
        self.output = input
        return self.output

    def backward(self, retain_graph=True):
        self.loss.backward(retain_graph=retain_graph)
        return self.loss
```

注意：

尽管它命名为内容损失，但它实际上不是真正的 PyTorch 损失函数。如果你想要定义内容损失函数作为 PyTorch 损失，你需要创建 PyTorch 自动求导函数并在反向传播中重新计算/实现梯度。

（6）风格损失

我们首先需要定义计算G_{XL}的模型。F_{XL}转为K×N的矩阵，设为$\hat{F}_{XL}$，其中K为特征图在L层的数量，N为特征图向量F_{XL}^{k}的长度。$\hat{F}_{XL}$的第k行为F_{XL}^{k}。有$\hat{F}_{XL} \cdot \hat{F}_{XL}^{T} = G_{XL}$。鉴于此，实现模型变得很简单：

```
class GramMatrix(nn.Module):

    def forward(self, input):
        a, b, c, d = input.size()  # a=batch size(=1)
        # b=number of feature maps
        # (c,d)=dimensions of a f. map (N=c*d)

        features = input.view(a * b, c * d)  # resise F_XL into \hat F_XL

        G = torch.mm(features, features.t())  # compute the gram product

        # we 'normalize' the values of the gram matrix
        # by dividing by the number of element in each feature maps.
        return G.div(a * b * c * d)
```

特征图的维度越长，gram 矩阵的值越大。因此，如果我们不用N归一化，第一层计算的损失（池化层之前）将决定梯度下降。我们并不想如此，最深的层拥有最有趣的风格特征。

我们定义风格损失的方式类似于内容损失，但我们需要增加 gramMatrix 作为参数：

```
class StyleLoss(nn.Module):

    def __init__(self, target, weight):
        super(StyleLoss, self).__init__()
        self.target = target.detach() * weight
        self.weight = weight
        self.gram = GramMatrix()
        self.criterion = nn.MSELoss()

    def forward(self, input):
        self.output = input.clone()
        self.G = self.gram(input)
        self.G.mul_(self.weight)
        self.loss = self.criterion(self.G, self.target)
        return self.output

    def backward(self, retain_graph=True):
        self.loss.backward(retain_graph=retain_graph)
        return self.loss
```

（7）载入神经网络

我们使用 VGG19。只需要它特征提取部分的网络：

```
cnn = models.vgg19(pretrained=True).features

# move it to the GPU if possible:
if use_cuda:
    cnn = cnn.cuda()
```

我们想要在我们的网络相关层中添加风格和内容损失作为附加的透明层。为此，我们创建一个新的序贯模型，在模型添加 VGG19 模型和我们的损失模型：

```
# desired depth layers to compute style/content losses :
content_layers_default = ['conv_4']
style_layers_default = ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']


def get_style_model_and_losses(cnn, style_img, content_img,
                               style_weight=1000, content_weight=1,
                               content_layers=content_layers_default,
                               style_layers=style_layers_default):
    cnn = copy.deepcopy(cnn)

    # just in order to have an iterable access to or list of content/syle
    # losses
    content_losses = []
    style_losses = []

    model = nn.Sequential()  # the new Sequential module network
    gram = GramMatrix()  # we need a gram module in order to compute style targets

    # move these modules to the GPU if possible:
    if use_cuda:
        model = model.cuda()
        gram = gram.cuda()

    i = 1
    for layer in list(cnn):
        if isinstance(layer, nn.Conv2d):
            name = "conv_" + str(i)
            model.add_module(name, layer)

            if name in content_layers:
                # add content loss:
                target = model(content_img).clone()
                content_loss = ContentLoss(target, content_weight)
                model.add_module("content_loss_" + str(i), content_loss)
                content_losses.append(content_loss)

            if name in style_layers:
                # add style loss:
                target_feature = model(style_img).clone()
                target_feature_gram = gram(target_feature)
                style_loss = StyleLoss(target_feature_gram, style_weight)
                model.add_module("style_loss_" + str(i), style_loss)
                style_losses.append(style_loss)
```

```
        if isinstance(layer, nn.ReLU):
            name = "relu_" + str(i)
            model.add_module(name, layer)

            if name in content_layers:
                # add content loss:
                target = model(content_img).clone()
                content_loss = ContentLoss(target, content_weight)
                model.add_module("content_loss_" + str(i), content_loss)
                content_losses.append(content_loss)

            if name in style_layers:
                # add style loss:
                target_feature = model(style_img).clone()
                target_feature_gram = gram(target_feature)
                style_loss = StyleLoss(target_feature_gram, style_weight)
                model.add_module("style_loss_" + str(i), style_loss)
                style_losses.append(style_loss)

            i += 1

        if isinstance(layer, nn.MaxPool2d):
            name = "pool_" + str(i)
            model.add_module(name, layer)  # ***

    return model, style_losses, content_losses
```

注意：

论文作者建议把最大池化改为平均池化。AlexNet 相对于 VGG19 是一个小网络，论文中使用它测试时效果没什么区别。但你如果想按照论文所述去替换，可以用下面几行内容：

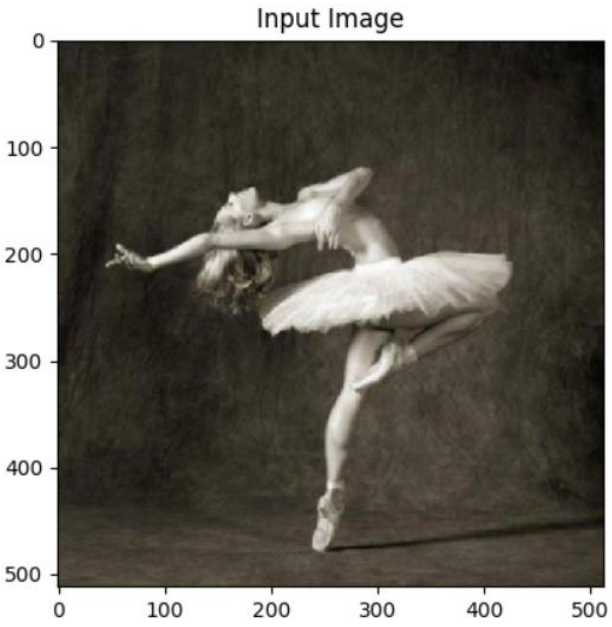
```
# avgpool = nn.AvgPool2d(kernel_size=layer.kernel_size,
#                        stride=layer.stride, padding = layer.padding)
# model.add_module(name,avgpool)
```

（8）输入图像

为了简化，还是让图像的维度一致。图像可以是白噪声或者只是内容图像的 copy：

```
input_img = content_img.clone()
# if you want to use a white noise instead uncomment the below line:
# input_img = Variable(torch.randn(content_img.data.size())).type(dtype)

# add the original input image to the figure:
plt.figure()
imshow(input_img.data, title='Input Image')
```

结果：



（9）梯度下降

使用 L-BFGS 算法。不像训练一个网络，我们想训练我们的输入图像，来最小化内容和风格损失。我们创建 L-BFGS 优化器，传入我们的图像作为需要优化的变量。但是 optim.LBFGS 的第一参数为含梯度的 PyTorch 变量 variable 列表。我们的输入是变量，但没有梯度。我们利用输入图像构建一个 Parameter 对象来使其具有梯度，这样就可以输入优化器了：

```
def get_input_param_optimizer(input_img):
    # this line to show that input is a parameter that requires a gradient
    input_param = nn.Parameter(input_img.data)
    optimizer = optim.LBFGS([input_param])
    return input_param, optimizer
```

最后一步：

循环，每次输入新的图像来计算新的损失，我们利用反向传播自动计算损失的梯度并更新变量。优化器需要一个“closure”作为参数：一个重新评估模型并返回损失的函数。

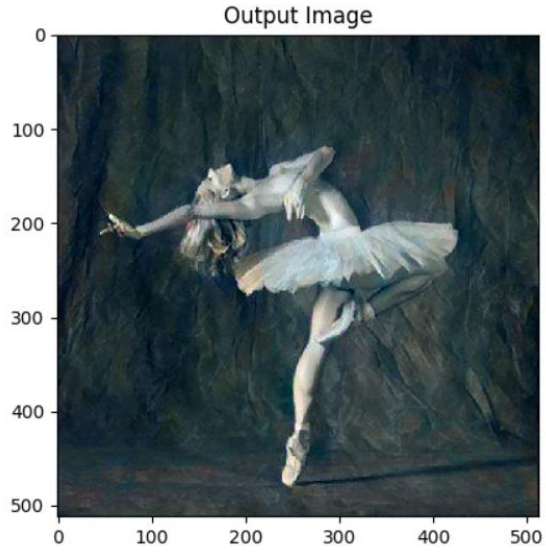
这有个问题，优化后的图像像素值在−∞到+∞之间，因此每次都需要调整图像像素值在 0 到 1 之间。

```
def run_style_transfer(cnn, content_img, style_img, input_img, num_steps=300,
                       style_weight=1000, content_weight=1):
    """Run the style transfer."""
    print('Building the style transfer model..')
    model, style_losses, content_losses = get_style_model_and_losses(cnn,
        style_img, content_img, style_weight, content_weight)
    input_param, optimizer = get_input_param_optimizer(input_img)

    print('Optimizing..')
    run = [0]
    while run[0] <= num_steps:

        def closure():
            # correct the values of updated input image
            input_param.data.clamp_(0, 1)

            optimizer.zero_grad()
            model(input_param)
            style_score = 0
            content_score = 0

            for sl in style_losses:
                style_score += sl.backward()
            for cl in content_losses:
                content_score += cl.backward()

            run[0] += 1
            if run[0] % 50 == 0:
                print("run {}:".format(run))
                print('Style Loss : {:4f} Content Loss: {:4f}'.format(
                    style_score.data[0], content_score.data[0]))
                print()

            return style_score + content_score

        optimizer.step(closure)

    # a last correction...
    input_param.data.clamp_(0, 1)

    return input_param.data
```

最后，运行这个算法：

```
output = run_style_transfer(cnn, content_img, style_img, input_img)

plt.figure()
imshow(output, title='Output Image')

# sphinx_gallery_thumbnail_number = 4
plt.ioff()
plt.show()
```

结果：




```
Building the style transfer model..
Optimizing..
run [50]:
Style Loss : 0.173669 Content Loss: 0.474930

run [100]:
Style Loss : 0.043105 Content Loss: 0.341928

run [150]:
Style Loss : 0.035798 Content Loss: 0.315916

run [200]:
Style Loss : 0.031996 Content Loss: 0.306762

run [250]:
Style Loss : 0.030744 Content Loss: 0.302271

run [300]:
Style Loss : 0.030627 Content Loss: 0.299732
```