# 实验报告

## 实验过程及结果

### 基本原理

原理很简单：我们定义了两个距离，一个用于内容 (*DC*​） 和一个用于样式 *DS*​)。*DC*​衡量内容的不同程度 位于两个图像之间，而*DS*​衡量风格的不同程度 在两个图像之间。然后，我们获取第三张图像，输入，然后 转换它以最小化其内容距离 content-image 及其与 style-image 的 style-distance 的距离。

### 导入软件包和选择设备

以下是实现神经传输所需的软件包列表。

* Torch，torch.nn ，numpy （使用 PyTorch 的神经网络不可或缺的软件包）
* torch.optim（高效的梯度下降）
* PIL、PIL.Image 、matplotlib.pyplot（加载和显示图片）
* torchvision.transforms（将PIL图像转换为张量）
* torchvision.models（训练或加载预训练模型）
* copy（深度拷贝模型;系统包）

代码如下：

|  |
| --- |
| import torch  import torch.nn as nn  import torch.nn.functional as F  import torch.optim as optim  from PIL import Image  import matplotlib.pyplot as plt  import torchvision.transforms as transforms  from torchvision.models import vgg19, VGG19\_Weights  import copy |

接下来，我们需要选择要在哪个设备上运行网络并导入 内容和样式图像。

|  |
| --- |
| device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  torch.set\_default\_device(device) |

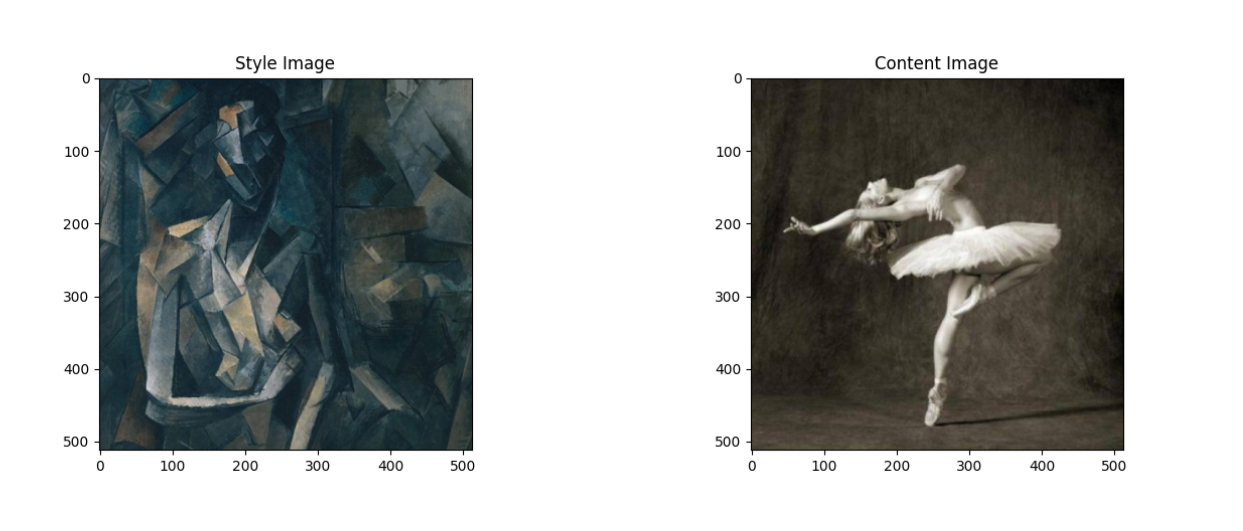
## 加载图像

现在我们将导入样式和内容图像。原始 PIL 映像的值介于 0 和 255 之间，但当 转换为火炬张量，它们的值被转换为介于 0 和 1。还需要调整图像大小以具有相同的尺寸。 需要注意的一个重要细节是，来自 Torch 库使用从 0 到 1 的张量值进行训练。如果你 尝试向网络提供 0 到 255 张量图像，然后激活 特征图将无法感知预期的内容和样式。 但是，来自 Caffe 库的预训练网络使用 0 进行训练 到 255 张量图像。

|  |
| --- |
| # desired size of the output image  imsize = 512 if torch.cuda.is\_available() else 128 # use small size if no GPU  loader = transforms.Compose([  transforms.Resize(imsize), # scale imported image  transforms.ToTensor()]) # transform it into a torch tensor  def image\_loader(image\_name):  image = Image.open(image\_name)  # fake batch dimension required to fit network's input dimensions  image = loader(image).unsqueeze(0)  return image.to(device, torch.float)  style\_img = image\_loader("./data/images/neural-style/picasso.jpg")  content\_img = image\_loader("./data/images/neural-style/dancing.jpg")  assert style\_img.size() == content\_img.size(), \  "we need to import style and content images of the same size" |

现在，让我们创建一个函数，通过重新转换 将其复制到 PIL 格式，并使用 显示副本。我们将尝试显示内容和样式图像 以确保它们被正确导入。

|  |
| --- |
| unloader = transforms.ToPILImage() # reconvert into PIL image  plt.ion()  def imshow(tensor, title=None):  image = tensor.cpu().clone() # we clone the tensor to not do changes on it  image = image.squeeze(0) # 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, title='Style Image')  plt.figure()  imshow(content\_img, title='Content Image') |



### 损失函数

#### 内容损失

内容丢失是一个函数，表示单个图层的内容距离。该函数采用该功能地图FXL​的图层L在网络处理输入中X并返回 加权内容距离wCL​.DCL​(X，C)图像之间X和内容图像C.内容图像的特征图（FCL​）必须是通过已知函数来计算内容距离。我们 将此函数实现为 Torch 模块，其构造函数采用FCL​作为输入。距离||FXL​−FCL|||2在两组特征图之间，可以使用nn.MSELoss计算。

我们将在卷积之后直接添加此内容损失模块用于计算内容距离的图层。这边每次向网络馈送输入图像时，内容损失都会在所需图层上计算，并且由于自动渐变，所有将计算梯度。现在，为了制作内容损失层透明 我们必须定义一个计算内容的方法 loss，然后返回图层的输入。计算出的损失保存为forward模块的参数。

|  |
| --- |
| class ContentLoss(nn.Module):  def \_\_init\_\_(self, target,):  super(ContentLoss, self).\_\_init\_\_()  # we 'detach' the target content from the tree used  # 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.target = target.detach()  def forward(self, input):  self.loss = F.mse\_loss(input, self.target)  return input |

#### 样式损失

样式丢失模块的实现方式与内容丢失类似 模块。它将在 计算该图层的样式损失的网络。为了 计算样式损失，我们需要计算 gram 矩阵GXL​.一克 矩阵是给定矩阵乘以其转置的结果 矩阵。在此应用程序中，给定的矩阵是 功能图FXL​的图层L.FXL​被重塑成F^XL​一个KxN矩阵，其中K是图层上特征图的数量L和N是任何矢量化特征图的长度FXLk​.例如，第一行F^XL​对应于第一个矢量化特征图F1X L​。

最后，必须通过将每个元素除以 矩阵中的元素总数。这种规范化是 抵消以下事实F^XL​矩阵具有大N尺寸产量 Gram 矩阵中的值较大。这些较大的值将导致 第一层（在池化层之前）在 梯度下降。样式特征往往位于 网络，所以这个规范化步骤是至关重要的。

|  |
| --- |
| def gram\_matrix(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) # resize 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) |

现在，样式丢失模块看起来几乎与内容丢失完全相同 模块。样式距离也是使用均方计算的 之间的误差*GXL*​和*GS* *L*

### 导入模型

现在我们需要导入一个预训练的神经网络。我们将使用 VGG19网络，如论文中使用的网络。

|  |
| --- |
| cnn = vgg19(weights=VGG19\_Weights.DEFAULT).features.eval() |

此外，VGG网络在每个通道的图像上进行训练归一化为 mean=[0.485， 0.456， 0.406] 和 std=[0.229， 0.224， 0.225]。在将图像发送到网络之前，我们将使用它们对图像进行归一化。

|  |
| --- |
| cnn\_normalization\_mean = torch.tensor([0.485, 0.456, 0.406])  cnn\_normalization\_std = torch.tensor([0.229, 0.224, 0.225])  # create a module to normalize input image so we can easily put it in a  # ``nn.Sequential``  class Normalization(nn.Module):  def \_\_init\_\_(self, mean, std):  super(Normalization, self).\_\_init\_\_()  # .view the mean and std to make them [C x 1 x 1] so that they can  # directly work with image Tensor of shape [B x C x H x W].  # B is batch size. C is number of channels. H is height and W is width.  self.mean = torch.tensor(mean).view(-1, 1, 1)  self.std = torch.tensor(std).view(-1, 1, 1)  def forward(self, img):  # normalize ``img``  return (img - self.mean) / self.std |

模块包含子模块的有序列表。为 instance，包含一个按正确深度顺序排列的序列 。我们需要添加我们的 卷积后立即出现内容损失和样式损失层 他们正在检测的层。为此，我们必须创建一个新模块，该模块正确插入了内容丢失和样式丢失模块。

|  |
| --- |
| # 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, normalization\_mean, normalization\_std,  style\_img, content\_img,  content\_layers=content\_layers\_default,  style\_layers=style\_layers\_default):  # normalization module  normalization = Normalization(normalization\_mean, normalization\_std)  # just in order to have an iterable access to or list of content/style  # losses  content\_losses = []  style\_losses = []  # assuming that ``cnn`` is a ``nn.Sequential``, so we make a new ``nn.Sequential``  # to put in modules that are supposed to be activated sequentially  model = nn.Sequential(normalization)  i = 0 # increment every time we see a conv  for layer in cnn.children():  if isinstance(layer, nn.Conv2d):  i += 1  name = 'conv\_{}'.format(i)  elif isinstance(layer, nn.ReLU):  name = 'relu\_{}'.format(i)  # The in-place version doesn't play very nicely with the ``ContentLoss``  # and ``StyleLoss`` we insert below. So we replace with out-of-place  # ones here.  layer = nn.ReLU(inplace=False)  elif isinstance(layer, nn.MaxPool2d):  name = 'pool\_{}'.format(i)  elif isinstance(layer, nn.BatchNorm2d):  name = 'bn\_{}'.format(i)  else:  raise RuntimeError('Unrecognized layer: {}'.format(layer.\_\_class\_\_.\_\_name\_\_))  model.add\_module(name, layer)  if name in content\_layers:  # add content loss:  target = model(content\_img).detach()  content\_loss = ContentLoss(target)  model.add\_module("content\_loss\_{}".format(i), content\_loss)  content\_losses.append(content\_loss)  if name in style\_layers:  # add style loss:  target\_feature = model(style\_img).detach()  style\_loss = StyleLoss(target\_feature)  model.add\_module("style\_loss\_{}".format(i), style\_loss)  style\_losses.append(style\_loss)  # now we trim off the layers after the last content and style losses  for i in range(len(model) - 1, -1, -1):  if isinstance(model[i], ContentLoss) or isinstance(model[i], StyleLoss):  break  model = model[:(i + 1)]  return model, style\_losses, content\_losses |

接下来，选择输入图像。可以使用内容图像的副本或白噪声。

|  |
| --- |
| input\_img = content\_img.clone()  # if you want to use white noise by using the following code:  #  # ::  #  # input\_img = torch.randn(content\_img.data.size())  # add the original input image to the figure:  plt.figure()  imshow(input\_img, title='Input Image') |

### 梯度下降

正如该算法的作者 Leon Gatys [在这里](https://discuss.pytorch.org/t/pytorch-tutorial-for-neural-transfert-of-artistic-style/336/20?u=alexis-jacq)建议的那样，我们将使用 L-BFGS 算法来运行梯度下降。与训练网络不同， 我们想训练输入图像，以最小化内容/样式 损失。我们将创建一个 PyTorch L-BFGS 优化器并传递 我们的图像以它为张量进行优化。

|  |
| --- |
| def get\_input\_optimizer(input\_img):  # this line to show that input is a parameter that requires a gradient  optimizer = optim.LBFGS([input\_img])  return optimizer |

最后，我们必须定义一个执行神经转移的函数。为 每次网络迭代，它都会被馈送到更新的输入并计算 新的损失。我们将运行每个 loss 模块的方法 动态计算它们的梯度。优化器需要“闭包” 函数，该函数重新计算模块并返回损失。

我们还有最后一个制约因素需要解决。网络可能会尝试 使用超过 0 到 1 张量范围的值优化输入 图像。我们可以通过将输入值更正为 每次运行网络时，介于 0 到 1 之间。

|  |
| --- |
| def run\_style\_transfer(cnn, normalization\_mean, normalization\_std,  content\_img, style\_img, input\_img, num\_steps=300,  style\_weight=1000000, content\_weight=1):  """Run the style transfer."""  print('Building the style transfer model..')  model, style\_losses, content\_losses = get\_style\_model\_and\_losses(cnn,  normalization\_mean, normalization\_std, style\_img, content\_img)  # We want to optimize the input and not the model parameters so we  # update all the requires\_grad fields accordingly  input\_img.requires\_grad\_(True)  # We also put the model in evaluation mode, so that specific layers  # such as dropout or batch normalization layers behave correctly.  model.eval()  model.requires\_grad\_(False)  optimizer = get\_input\_optimizer(input\_img)  print('Optimizing..')  run = [0]  while run[0] <= num\_steps:  def closure():  # correct the values of updated input image  with torch.no\_grad():  input\_img.clamp\_(0, 1)  optimizer.zero\_grad()  model(input\_img)  style\_score = 0  content\_score = 0  for sl in style\_losses:  style\_score += sl.loss  for cl in content\_losses:  content\_score += cl.loss  style\_score \*= style\_weight  content\_score \*= content\_weight  loss = style\_score + content\_score  loss.backward()  run[0] += 1  if run[0] % 50 == 0:  print("run {}:".format(run))  print('Style Loss : {:4f} Content Loss: {:4f}'.format(  style\_score.item(), content\_score.item()))  print()  return style\_score + content\_score  optimizer.step(closure)  # a last correction...  with torch.no\_grad():  input\_img.clamp\_(0, 1)  return input\_img |

最后，运行算法

|  |
| --- |
| output = run\_style\_transfer(cnn, cnn\_normalization\_mean, cnn\_normalization\_std,  content\_img, style\_img, input\_img)  plt.figure()  imshow(output, title='Output Image')  # sphinx\_gallery\_thumbnail\_number = 4  plt.ioff()  plt.show() |

不同参数条件下的实验结果如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **content\_weight** | **style\_weight** | **结果** |  |
| 100 | 1 |  | run [50]:  Style Loss : 0.002930 Content Loss: 0.000000  run [100]:  Style Loss : 0.002930 Content Loss: 0.000000  run [150]:  Style Loss : 0.002930 Content Loss: 0.000000  run [200]:  Style Loss : 0.002930 Content Loss: 0.000000  run [250]:  Style Loss : 0.002930 Content Loss: 0.000000  run [300]:  Style Loss : 0.002930 Content Loss: 0.000000 |
| 10 | 1 |  | run [50]:  Style Loss : 0.002930 Content Loss: 0.000000  run [100]:  Style Loss : 0.002930 Content Loss: 0.000000  run [150]:  Style Loss : 0.002930 Content Loss: 0.000000  run [200]:  Style Loss : 0.002930 Content Loss: 0.000000  run [250]:  Style Loss : 0.002930 Content Loss: 0.000000  run [300]:  Style Loss : 0.002930 Content Loss: 0.000000 |
| 1 | 1 |  | run [50]:  Style Loss : 0.002926 Content Loss: 0.000002  run [100]:  Style Loss : 0.002925 Content Loss: 0.000002  run [150]:  Style Loss : 0.002925 Content Loss: 0.000003  run [200]:  Style Loss : 0.002925 Content Loss: 0.000003  run [250]:  Style Loss : 0.002925 Content Loss: 0.000003  run [300]:  Style Loss : 0.002925 Content Loss: 0.000003 |
| 1 | 10 |  | run [50]:  Style Loss : 0.028852 Content Loss: 0.000224  run [100]:  Style Loss : 0.028815 Content Loss: 0.000242  run [150]:  Style Loss : 0.028796 Content Loss: 0.000251  run [200]:  Style Loss : 0.028782 Content Loss: 0.000257  run [250]:  Style Loss : 0.028772 Content Loss: 0.000262  run [300]:  Style Loss : 0.028764 Content Loss: 0.000265 |
| 1 | 100 |  | run [50]:  Style Loss : 0.250741 Content Loss: 0.020334  run [100]:  Style Loss : 0.248304 Content Loss: 0.021397  run [150]:  Style Loss : 0.247315 Content Loss: 0.021799  run [200]:  Style Loss : 0.246727 Content Loss: 0.022007  run [250]:  Style Loss : 0.246228 Content Loss: 0.022205  run [300]:  Style Loss : 0.245862 Content Loss: 0.022344 |
| 1 | 1000 |  | run [50]:  Style Loss : 0.796904 Content Loss: 0.722348  run [100]:  Style Loss : 0.735511 Content Loss: 0.739859  run [150]:  Style Loss : 0.717615 Content Loss: 0.740804  run [200]:  Style Loss : 0.705029 Content Loss: 0.741904  run [250]:  Style Loss : 0.695135 Content Loss: 0.743520  run [300]:  Style Loss : 0.688980 Content Loss: 0.742540 |
| 1 | 10000 |  | run [50]:  Style Loss : 0.909527 Content Loss: 2.695061  run [100]:  Style Loss : 0.804986 Content Loss: 2.438284  run [150]:  Style Loss : 0.777571 Content Loss: 2.399016  run [200]:  Style Loss : 0.773412 Content Loss: 2.373012  run [250]:  Style Loss : 0.767679 Content Loss: 2.356466  run [300]:  Style Loss : 0.767772 Content Loss: 2.345774 |
| 1 | 100000 |  | run [50]:  Style Loss : 5.587721  Content Loss: 11.677505  run [100]:  Style Loss : 2.166567  Content Loss: 7.147610  run [150]:  Style Loss : 1.495127  Content Loss: 5.516419  run [200]:  Style Loss : 1.116613  Content Loss: 4.836705  run [250]:  Style Loss : 0.996438  Content Loss: 4.497494  run [300]:  Style Loss : 0.948231  Content Loss: 4.336929 |

## 与另一版代码的区别

1. 选取的优化器不同

在另一版代码中选择的是Adam优化器，而此份代码中使用的是LBFGS优化器。Adam（Adaptive Moment Estimation）优化器和LBFGS（Limited-memory Broyden–Fletcher–Goldfarb–Shanno）优化器都是用于训练神经网络和其他机器学习模型的常见优化算法，它们在原理、适用场景以及特点上有一定的不同：

Adam优化器：

1. Adam 是一种基于梯度的一阶矩估计（即动量，也就是过去梯度的指数移动平均）和二阶矩估计（即梯度平方的指数移动平均，用于计算自适应学习率）的优化算法。这使得它能够自动调整每个参数的学习率，对不同的参数进行差异化更新。
2. 由于使用了二阶矩估计来调整学习率，Adam 对于具有稀疏梯度或者非均匀空间中的梯度有很好的表现，尤其适合处理大规模数据集和深度神经网络时的梯度消失或爆炸问题。
3. Adam 只需要设置较少的超参数，如学习率（lr）、动量项系数（β1）、二次矩估计项系数（β2）以及修正因子（ε），不需要像 LBFGS 那样存储历史梯度

L-BFGS优化器：

1. LBFGS 是一个基于拟牛顿法的迭代优化算法，属于二阶优化器家族的一员。它利用的是梯度的近似Hessian矩阵逆（不直接计算Hessian矩阵，而是通过有限内存技术存储最近的梯度信息来近似）来进行方向搜索。
2. 由于考虑了目标函数的曲率信息，LBFGS 在接近极小点附近通常能获得更快的收敛速度，并且对于凸优化问题效果良好。
3. 与一阶方法相比，LBFGS 需要更多的内存来存储历史梯度和步长信息以构建Hessian逆的近似。对于大型神经网络，尤其是在GPU资源有限的情况下，这可能成为限制因素。
4. LBFGS 主要依赖于线搜索策略的参数（例如步长的选择规则），相比于Adam来说，虽然不需显式地设置动量项等参数，但其内部实现更复杂，且对初始学习率选择较为敏感。

在实践中，Adam 因其自适应性和较低的维护成本而广泛应用于深度学习领域；而LBFGS更多见于小型到中型规模的优化问题，尤其是当问题的局部特性对收敛至关重要时。在本次进行的风格迁移实验中， Adam 优化器因为其高效性和鲁棒可能相对更优。

1. 生成新图片方式不同：

另一版代码中使用的是根据输入的内容图片和样式图片从零生成一张新图片，或者说在一张新的空白图像（通常是随机噪声或均匀分布的像素值）基础上更新像素值，在保存内容图像的基础上反映风格特征。

在此版代码使用的是直接以内容图像为基底，在其上叠加或修改风格元素，生成具有目标风格的新图片，实现过程中同样会利用卷积神经网络提取内容图片特征，并通过优化过程调整内容图片的像素，但是目标不再是无中生有地创造一张新的结构化图像，而是改变现有图像的视觉表达以符合给定的风格特征。

这两种方法的主要区别在于生成流程的起点和保留原始内容的方式。前者更为通用，允许完全创造性地合成图像，但可能较难控制最终输出与内容图片的一致性；后者则能保证生成的图片内容不变，只是样式发生了变化，更适合应用于需要保留特定内容细节的应用场景。

1. TV损失

在另一版代码中额外设置了TV损失，TV损失的设置主要用于促进图像的空间连续性和边缘平滑性，减少不必要的高频变化，也就是减少图像中的噪声和突变。添加TV损失原因主要在于从零生成一张从零开始生成新图片的过程中，特别是在生成高分辨率图像时，由于模型需要自动生成每一个像素点，更容易产生视觉上的噪点和不连续边缘，为了确保生成的图像更加平滑自然，确实更有可能需要使用Total Variation (TV) 损失作为正则化项来优化生成器。

相比之下，以内容图片为基底生成图片时，模型往往已经有了较好的初始结构和细节，只需要在此基础上进行修改或增强，这时虽然也可能用到TV损失来优化图像质量，但原始内容的存在一定程度上已经降低了生成过程中的噪声问题，当然使用TV损失同样可以进行一定程度的优化。