Style_Transfer_Exercise

May 15, 2020

1 Style Transfer with Deep Neural Networks

In this notebook, we'll *recreate* a style transfer method that is outlined in the paper, Image Style Transfer Using Convolutional Neural Networks, by Gatys in PyTorch.

In this paper, style transfer uses the features found in the 19-layer VGG Network, which is comprised of a series of convolutional and pooling layers, and a few fully-connected layers. In the image below, the convolutional layers are named by stack and their order in the stack. Conv_1_1 is the first convolutional layer that an image is passed through, in the first stack. Conv_2_1 is the first convolutional layer in the *second* stack. The deepest convolutional layer in the network is conv_5_4.

1.0.1 Separating Style and Content

Style transfer relies on separating the content and style of an image. Given one content image and one style image, we aim to create a new, *target* image which should contain our desired content and style components: * objects and their arrangement are similar to that of the **content image** * style, colors, and textures are similar to that of the **style image**

An example is shown below, where the content image is of a cat, and the style image is of Hokusai's Great Wave. The generated target image still contains the cat but is stylized with the waves, blue and beige colors, and block print textures of the style image!

In this notebook, we'll use a pre-trained VGG19 Net to extract content or style features from a passed in image. We'll then formalize the idea of content and style *losses* and use those to iteratively update our target image until we get a result that we want. You are encouraged to use a style and content image of your own and share your work on Twitter with @udacity; we'd love to see what you come up with!

```
In [18]: # import resources
         %matplotlib inline

         from PIL import Image
         import matplotlib.pyplot as plt
         import numpy as np

         import torch
         import torch.optim as optim
         from torchvision import transforms, models
```

1.1 Load in VGG19 (features)

VGG19 is split into two portions: * `vgg19.features`, which are all the convolutional and pooling layers * `vgg19.classifier`, which are the three linear, classifier layers at the end

We only need the `features` portion, which we're going to load in and "freeze" the weights of, below.

In [19]:
```
# get the "features" portion of VGG19 (we will not need the "classifier" portion)
vgg = models.vgg19(pretrained=True).features

# freeze all VGG parameters since we're only optimizing the target image
for param in vgg.parameters():
    param.requires_grad_(False)
```

In [20]:
```
# move the model to GPU, if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

vgg.to(device)
```

Out[20]:
```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace)
  (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace)
  (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU(inplace)
  (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace)
  (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (24): ReLU(inplace)
  (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (26): ReLU(inplace)
  (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
  (29): ReLU(inplace)
  (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (31): ReLU(inplace)
  (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (33): ReLU(inplace)
  (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (35): ReLU(inplace)
  (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```

1.1.1 Load in Content and Style Images

You can load in any images you want! Below, we've provided a helper function for loading in any type and size of image. The `load_image` function also converts images to normalized Tensors.

Additionally, it will be easier to have smaller images and to squish the content and style images so that they are of the same size.

```
In [21]: def load_image(img_path, max_size=400, shape=None):
             ''' Load in and transform an image, making sure the image
                is <= 400 pixels in the x-y dims.'''

             image = Image.open(img_path).convert('RGB')

             # large images will slow down processing
             if max(image.size) > max_size:
                 size = max_size
             else:
                 size = max(image.size)

             if shape is not None:
                 size = shape

             in_transform = transforms.Compose([
                                transforms.Resize(size),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485, 0.456, 0.406),
                                                     (0.229, 0.224, 0.225))])

             # discard the transparent, alpha channel (that's the :3) and add the batch dimensio
             image = in_transform(image)[:3,:,:].unsqueeze(0)

             return image
```

Next, I'm loading in images by file name and forcing the style image to be the same size as the content image.

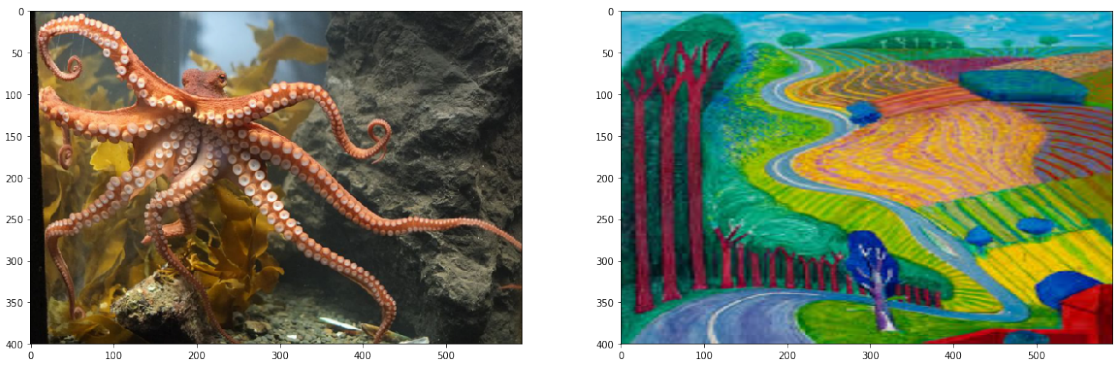
```
In [22]: # load in content and style image
         content = load_image('images/octopus.jpg').to(device)
```

```
        # Resize style to match content, makes code easier
        style = load_image('images/hockney.jpg', shape=content.shape[-2:]).to(device)
```

In [23]:
```
        # helper function for un-normalizing an image
        # and converting it from a Tensor image to a NumPy image for display
        def im_convert(tensor):
            """ Display a tensor as an image. """

            image = tensor.to("cpu").clone().detach()
            image = image.numpy().squeeze()
            image = image.transpose(1,2,0)
            image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
            image = image.clip(0, 1)

            return image
```

In [24]:
```
        # display the images
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
        # content and style ims side-by-side
        ax1.imshow(im_convert(content))
        ax2.imshow(im_convert(style))
```

Out[24]:
```
<matplotlib.image.AxesImage at 0x7f9b81f86278>
```



1.2 VGG19 Layers

To get the content and style representations of an image, we have to pass an image forward through the VGG19 network until we get to the desired layer(s) and then get the output from that layer.

In [25]:
```
        # print out VGG19 structure so you can see the names of various layers
        # print(vgg)
        print(vgg)
```

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace)
  (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace)
  (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU(inplace)
  (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace)
  (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (24): ReLU(inplace)
  (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (26): ReLU(inplace)
  (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace)
  (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (31): ReLU(inplace)
  (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (33): ReLU(inplace)
  (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (35): ReLU(inplace)
  (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```

1.3 Content and Style Features

TODO: complete the mapping of layer names to the names found in the paper for the *content representation* and the *style representation*. The first layer (0) to `conv1_1` has been done for you, below.

```
In [30]: def get_features(image, model, layers=None):
             """ Run an image forward through a model and get the features for
                 a set of layers. Default layers are for VGGNet matching Gatys et al (2016)
             """

             ## TODO: Complete mapping layer names of PyTorch's VGGNet to names from the paper
             ## Need the layers for the content and style representations of an image
             if layers is None:
                 layers = {'0': 'conv1_1',
                           '5': 'conv2_1',
                           '10': 'conv3_1',
                           '19': 'conv4_1',
                           '21': 'conv4_2',  ## content representation
                           '28': 'conv5_1'}


             ## -- do not need to change the code below this line -- ##
             features = {}
             x = image
             # model._modules is a dictionary holding each module in the model
             for name, layer in model._modules.items():
                 x = layer(x)
                 if name in layers:
                     features[layers[name]] = x

             return features
```

1.4 Gram Matrix

The output of every convolutional layer is a Tensor with dimensions associated with the `batch_size`, a depth, `d` and some height and width (`h`, `w`). The Gram matrix of a convolutional layer can be calculated as follows: * Get the depth, height, and width of a tensor using `batch_size, d, h, w = tensor.size()` * Reshape that tensor so that the spatial dimensions are flattened * Calculate the gram matrix by multiplying the reshaped tensor by it's transpose

Note: You can multiply two matrices using `torch.mm(matrix1, matrix2)`.

TODO: Complete the `gram_matrix` function.

```
In [31]: def gram_matrix(tensor):
             """ Calculate the Gram Matrix of a given tensor
                 Gram Matrix: https://en.wikipedia.org/wiki/Gramian_matrix
             """

             # get the batch_size, depth, height, and width of the Tensor
             _, d, h, w = tensor.size()
```

```
        # reshape so we're multiplying the features for each channel
        tensor = tensor.view(d, h * w)

        # calculate the gram matrix
        gram = torch.mm(tensor, tensor.t())

        return gram
```

1.5 Putting it all Together

Now that we've written functions for extracting features and computing the gram matrix of a given convolutional layer; let's put all these pieces together! We'll extract our features from our images and calculate the gram matrices for each layer in our style representation.

```
In [32]: # get content and style features only once before forming the target image
         content_features = get_features(content, vgg)
         style_features = get_features(style, vgg)

         # calculate the gram matrices for each layer of our style representation
         style_grams = {layer: gram_matrix(style_features[layer]) for layer in style_features}

         # create a third "target" image and prep it for change
         # it is a good idea to start of with the target as a copy of our *content* image
         # then iteratively change its style
         target = content.clone().requires_grad_(True).to(device)
```

1.6 Loss and Weights

Individual Layer Style Weights Below, you are given the option to weight the style representation at each relevant layer. It's suggested that you use a range between 0-1 to weight these layers. By weighting earlier layers (`conv1_1` and `conv2_1`) more, you can expect to get *larger* style artifacts in your resulting, target image. Should you choose to weight later layers, you'll get more emphasis on smaller features. This is because each layer is a different size and together they create a multi-scale style representation!

Content and Style Weight Just like in the paper, we define an alpha (`content_weight`) and a beta (`style_weight`). This ratio will affect how *stylized* your final image is. It's recommended that you leave the content_weight = 1 and set the style_weight to achieve the ratio you want.

```
In [33]: # weights for each style layer
         # weighting earlier layers more will result in *larger* style artifacts
         # notice we are excluding `conv4_2` our content representation
         style_weights = {'conv1_1': 1.,
                          'conv2_1': 0.8,
                          'conv3_1': 0.5,
                          'conv4_1': 0.3,
```

```
                      'conv5_1': 0.1}

        # you may choose to leave these as is
        content_weight = 1  # alpha
        style_weight = 1e6  # beta
```

1.7 Updating the Target & Calculating Losses

You'll decide on a number of steps for which to update your image, this is similar to the training loop that you've seen before, only we are changing our *target* image and nothing else about VGG19 or any other image. Therefore, the number of steps is really up to you to set! **I recommend using at least 2000 steps for good results.** But, you may want to start out with fewer steps if you are just testing out different weight values or experimenting with different images.

Inside the iteration loop, you'll calculate the content and style losses and update your target image, accordingly.

Content Loss The content loss will be the mean squared difference between the target and content features at layer `conv4_2`. This can be calculated as follows:

```
content_loss = torch.mean((target_features['conv4_2'] - content_features['conv4_2'])**2)
```

Style Loss The style loss is calculated in a similar way, only you have to iterate through a number of layers, specified by name in our dictionary `style_weights`. > You'll calculate the gram matrix for the target image, `target_gram` and style image `style_gram` at each of these layers and compare those gram matrices, calculating the `layer_style_loss`. > Later, you'll see that this value is normalized by the size of the layer.

Total Loss Finally, you'll create the total loss by adding up the style and content losses and weighting them with your specified alpha and beta!

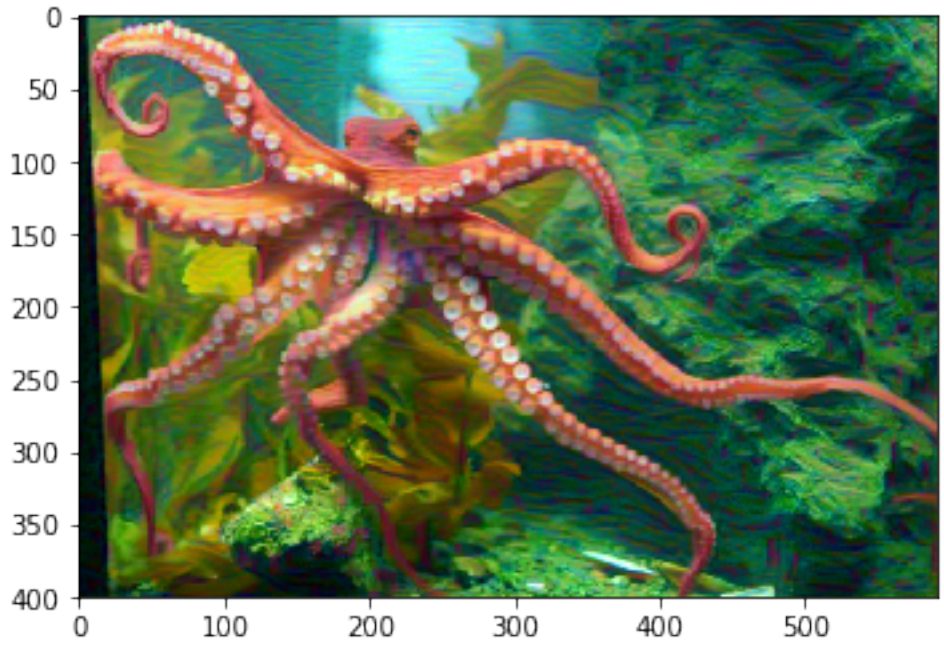
Intermittently, we'll print out this loss; don't be alarmed if the loss is very large. It takes some time for an image's style to change and you should focus on the appearance of your target image rather than any loss value. Still, you should see that this loss decreases over some number of iterations.

TODO: Define content, style, and total losses.

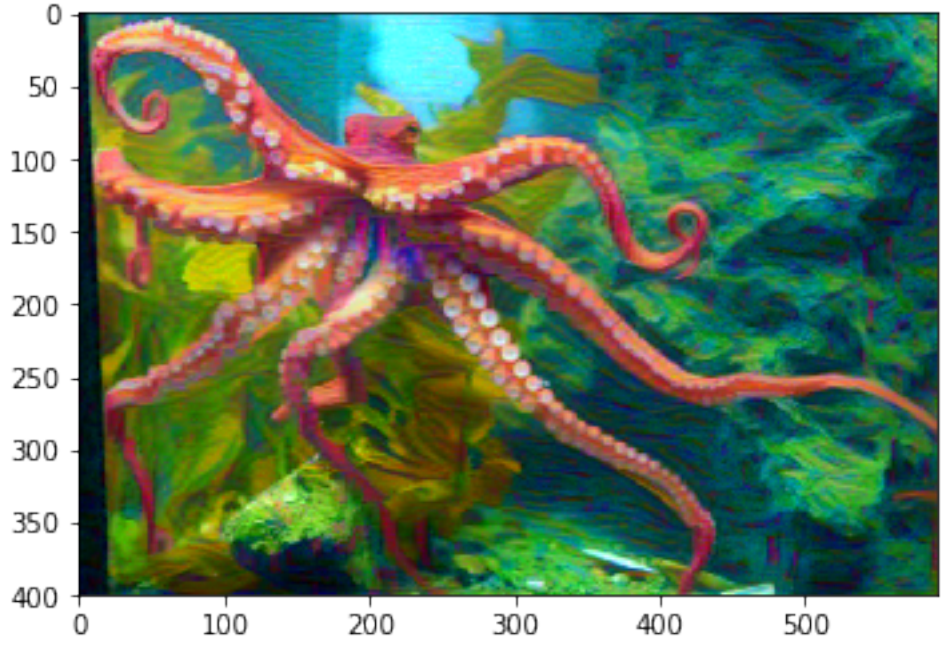
```
In [34]: # for displaying the target image, intermittently
         show_every = 400

         # iteration hyperparameters
         optimizer = optim.Adam([target], lr=0.003)
         steps = 2000  # decide how many iterations to update your image (5000)

         for ii in range(1, steps+1):

             ## TODO: get the features from your target image
             ## Then calculate the content loss
             target_features = get_features(target,vgg)
```

```
            content_loss = torch.mean((target_features['conv4_2'] - content_features['conv4_2'])

            # the style loss
            # initialize the style loss to 0
            style_loss = 0
            # iterate through each style layer and add to the style loss
            for layer in style_weights:
                # get the "target" style representation for the layer
                target_feature = target_features[layer]
                target_gram = gram_matrix(target_feature)
                _, d, h, w = target_feature.shape
                # get the "style" style representation
                style_gram = style_grams[layer]
                # the style loss for one layer, weighted appropriately
                layer_style_loss = style_weights[layer] * torch.mean((target_gram - style_gram)
                # add to the style loss
                style_loss += layer_style_loss / (d * h * w)


            ## TODO:  calculate the *total* loss
            total_loss = content_weight * content_loss + style_weight * style_loss

            ## -- do not need to change code, below -- ##
            # update your target image
            optimizer.zero_grad()
            total_loss.backward()
            optimizer.step()

            # display intermediate images and print the loss
            if  ii % show_every == 0:
                print('Total loss: ', total_loss.item())
                plt.imshow(im_convert(target))
                plt.show()
```

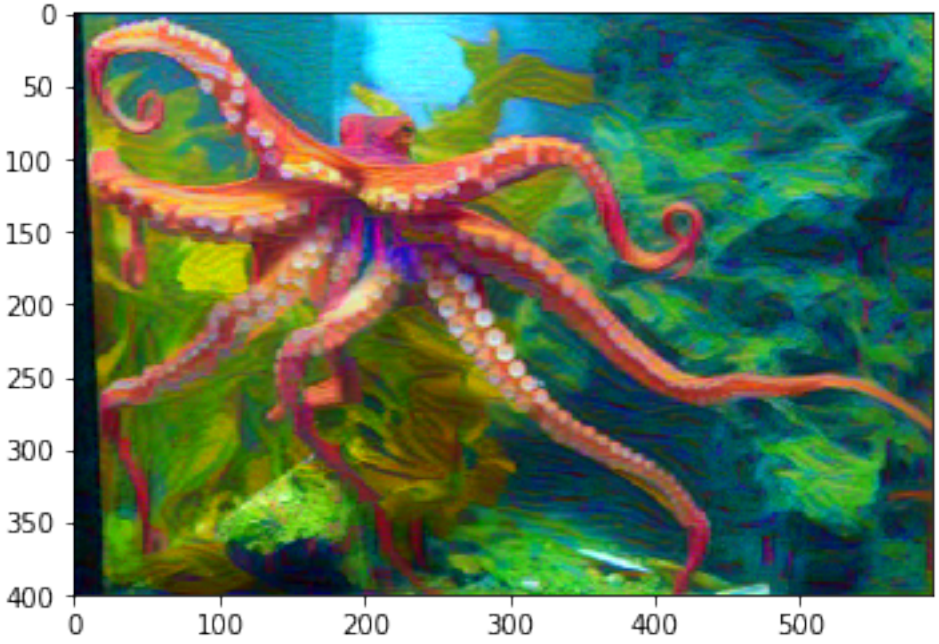
```
Total loss:  96250608.0
```



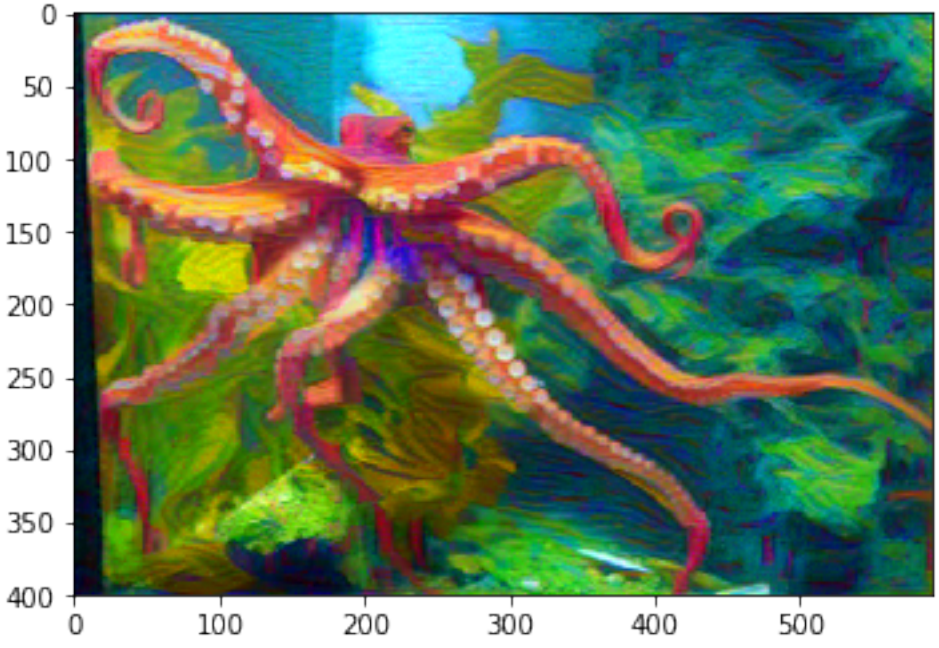

```
Total loss:  15237791.0
```



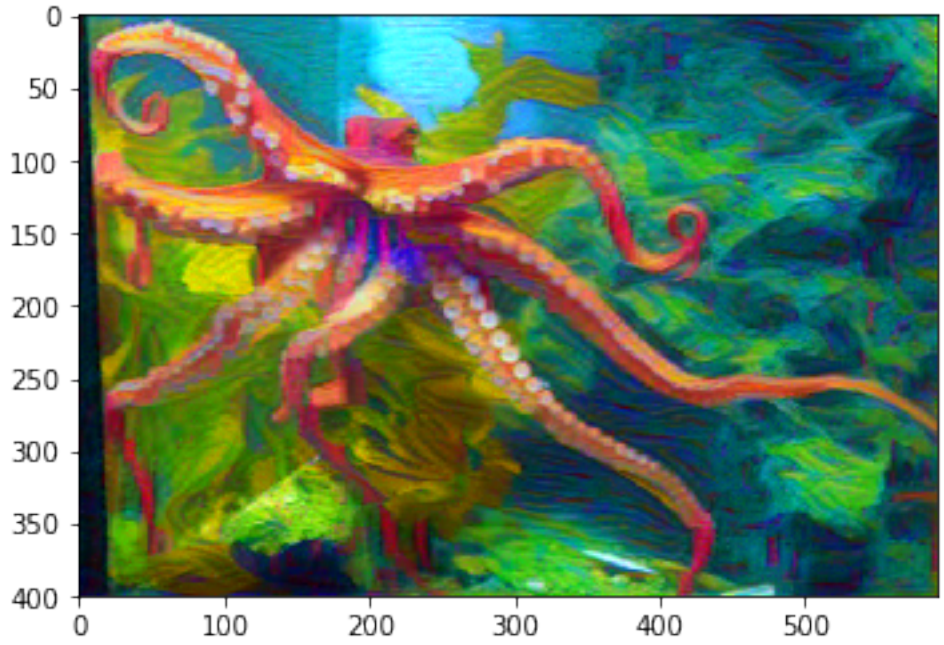
```
Total loss:  7416794.0
```




```
Total loss:  4610224.0
```



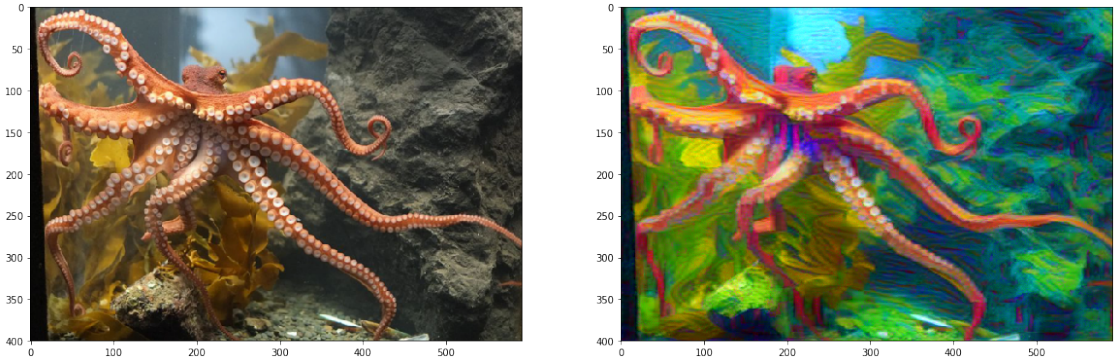
```
Total loss:  3087732.5
```



1.8 Display the Target Image

```
In [35]: # display content and final, target image
         fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
         ax1.imshow(im_convert(content))
         ax2.imshow(im_convert(target))
```

```
Out[35]: <matplotlib.image.AxesImage at 0x7f9b81cece48>
```



```
In [ ]:
```