In the previous chapter, we started exploring generative models using PyTorch. We built machine learning models that can generate text and music by training the models without supervision on text and music data, respectively. We will continue exploring generative modeling in this chapter by applying a similar methodology to image data.

We will mix different aspects of two different images, **A** and **B**, to generate a resultant image, **C**, that contains the content of image **A** and the style of image **B**. This task is also popularly known as **neural style transfer** because, in a way, we are transferring the style of image **B** to image **A** in order to achieve image **C**, as illustrated in the following figure:

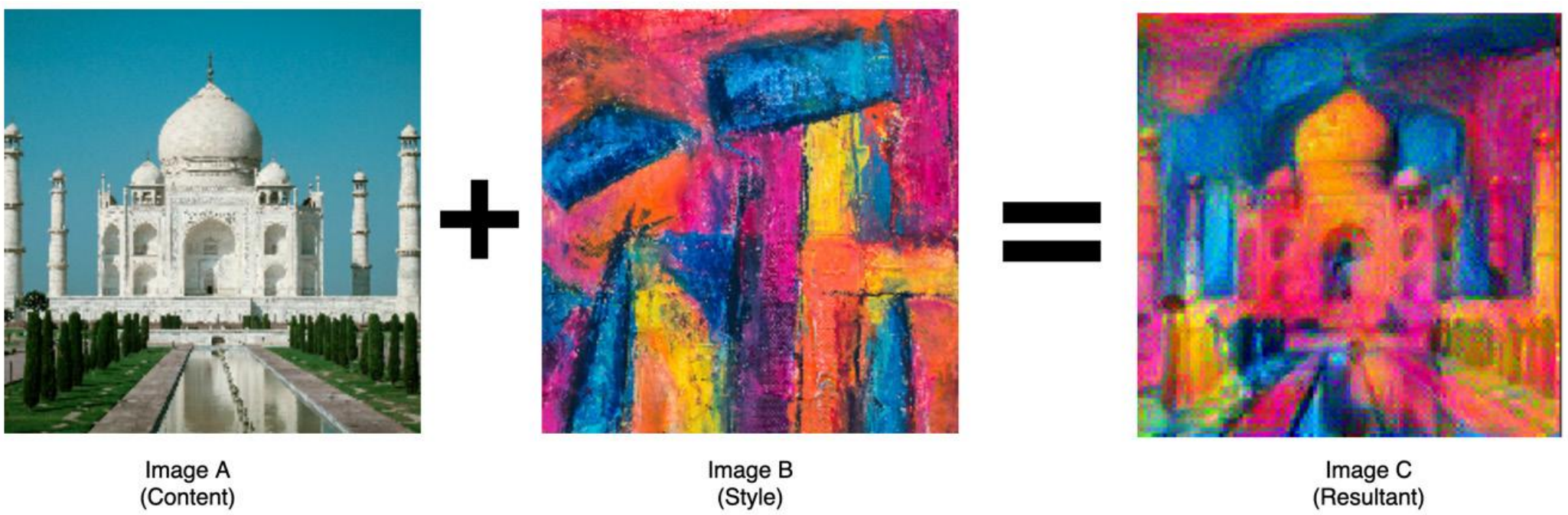

Figure 8.1 – Neural style transfer example

First, we will briefly discuss how to approach this problem and understand the idea behind achieving style transfer. Using PyTorch, we will then implement our own neural style transfer system and apply it to a pair of images. Through this implementation

exercise, we will also try to understand the effects of different parameters in the style transfer mechanism.

By the end of this chapter, you will understand the concepts behind neural style transfer and be able to build and test your own neural style transfer model using PyTorch.

This chapter covers the following topics:

- Understanding how to transfer style between images
- Implementing neural style transfer using PyTorch

Understanding how to transfer style between images

In *Chapter 3*, *Deep CNN Architectures*, we discussed **convolutional neural networks** (**CNNs**) in detail. CNNs are largely the most successful class of models when working with image data. We have seen how CNN-based architectures are among the best-performing architectures of neural networks on tasks such as image classification, object detection, and so on. One of the core reasons behind this success is the ability of convolutional layers to learn spatial representations.

For example, in a dog versus cat classifier, the CNN model is essentially able to capture the content of an image in its higher-level features, which helps it detect dog-specific features against cat-specific features. We will leverage this ability of an image classifier CNN to grasp the content of an image.

We know that VGG is a powerful image classification model, as discussed in *Chapter 3*, *Deep CNN Architectures*. We are going to use the convolutional part of the VGG model (excluding the linear layers) to extract content-related features from an image.

We know that each convolutional layer produces, say, *N* feature maps of dimensions *X*Y* each. For example, let's say we have a single channel (grayscale) input image of size (3,3) and a convolutional layer where the number of output channels (*M*) is 3, the

kernel size is (2,2) with a stride of (1,1), and there's no padding. This convolutional layer will produce 3 (*M*) feature maps each of size 2x2, hence *X*=2 and *Y*=2 in this case.

We can represent these *N* feature maps produced by the convolutional layer as a 2D matrix of size *N*M*, where *M=X*Y*. By defining the output of each convolutional layer as a 2D matrix, we can define a loss function that's attached to each convolutional layer. This loss function, called the **content loss**, is the squared loss between the expected and predicted outputs of the convolutional layers, as demonstrated in the following diagram, with *N*=3, *X*=2, and *Y*=2:

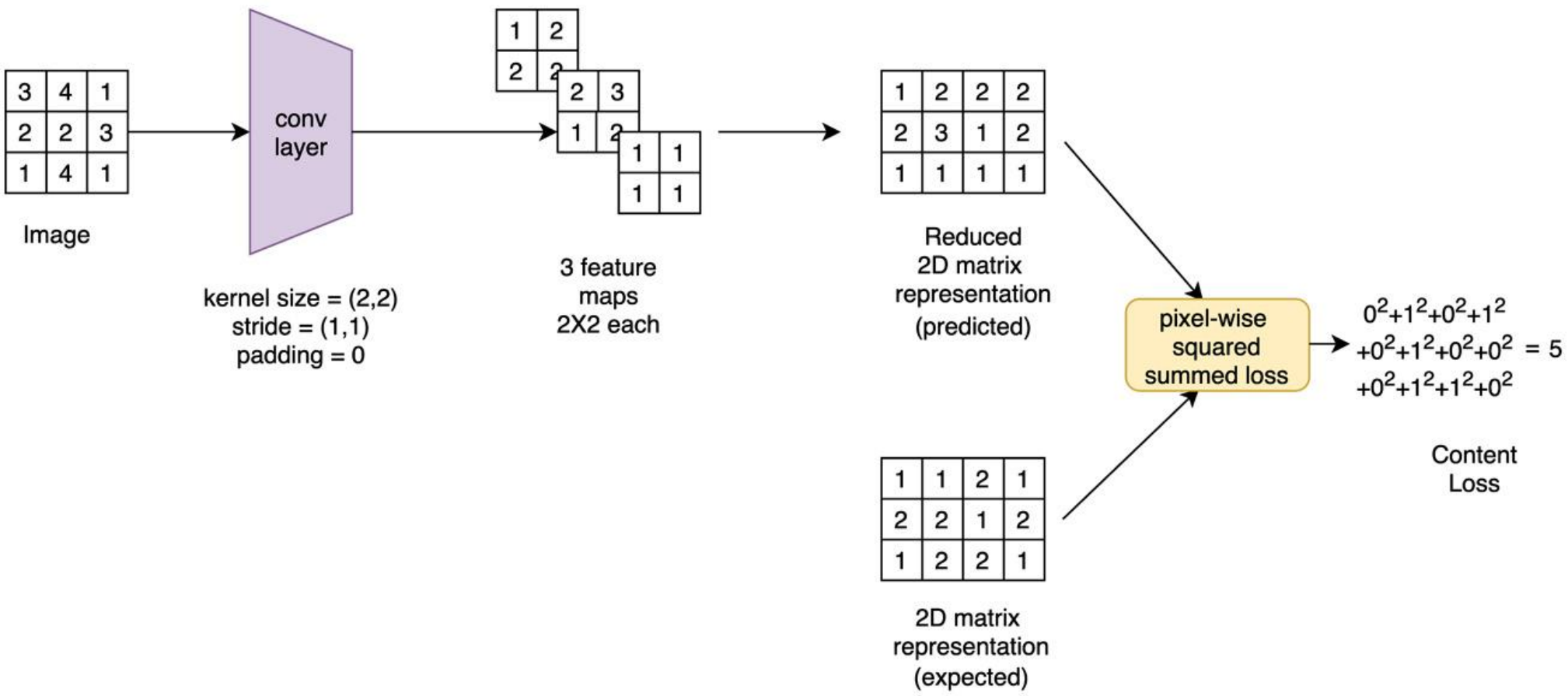

Figure 8. 2 – Content loss schematic

As we can see, the input image (image *C*, as per our notation in *Figure 8. 1*) in this example is transformed into **three feature maps** by the **convolutional (conv) layer.** These three feature maps, of size 2x2 each, are formatted into a 3x4 matrix. This matrix is compared with the expected output, which is obtained by passing image *A* (the content image) through the same flow. The pixel-wise squared summed loss is then calculated, which we call the **content loss.**

Now, for extracting style from an image, we will use gram matrices [8.1] derived from the inner product between the rows of the reduced 2D matrix representations, as demonstrated in the following diagram:

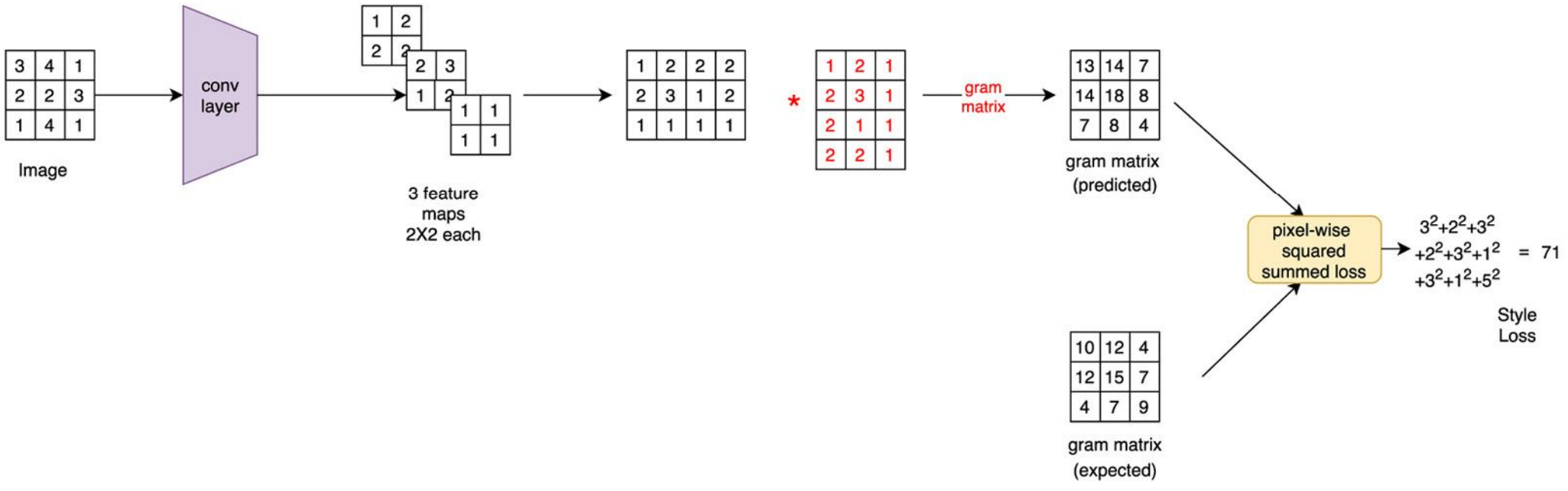

Figure 8. 3 – Style loss schematic

The **gram matrix** computation is the only extra step here compared to the content loss calculations. Also, as we can see, the output of the pixel-wise squared summed loss is quite a large number compared to the content loss. Hence, this number is normalized by dividing it by *N*X*Y*; that is, the number of feature maps (*N*) times the length (*X*) times the breadth (*Y*) of a feature map. This also helps standardize the **style loss** metric across different convolutional layers, which have a different *N*, *X*, and *Y*. Details of the implementation can be found in the original paper that introduced neural style transfer [8.2] .

Now that we understand the concept of content and style loss, let's take a look at how neural style transfer works, as follows:

1. For the given VGG (or any other CNN) network, we define which convolutional layers in the network should have a content loss attached to them. Repeat this exercise for style loss.
2. Once we have those lists, we pass the content image through the network and compute the expected convolutional outputs (2D matrices) at the convolutional layers where the content loss is to be calculated.
3. Next, we pass the style image through the network and compute the expected gram matrices at the convolutional layers. This is where the style loss is to be calculated, as demonstrated in the following diagram.

In the following diagram, for example, the content loss is to be calculated at the second and third convolutional layers, while the style loss is to be calculated at the second, third, and fifth convolutional layers:

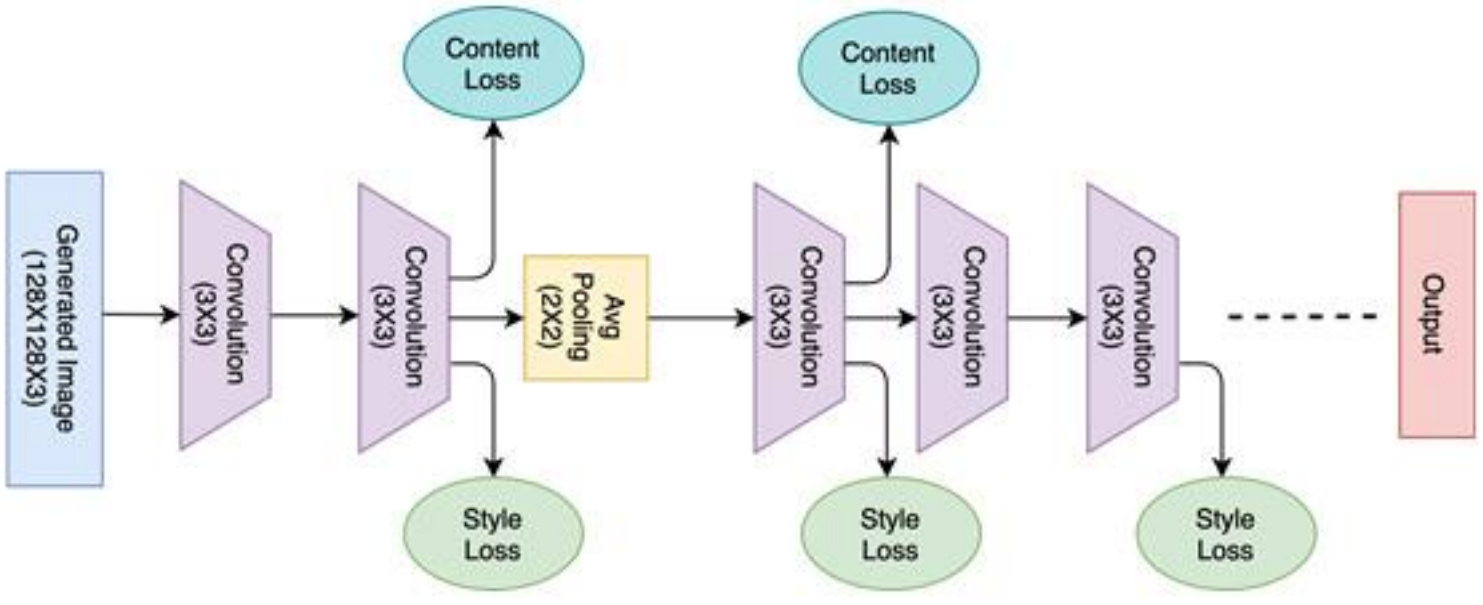

Figure 8. 4 – Style transfer architecture schematic

Now that we have the content and style targets at the decided convolutional layers, we are all set to generate an image that contains the content of the content image and the style of the style image.

For initialization, we can either use a random noise matrix as our starting point for the generated image, or directly use the content image to start with. We pass this image through the network and compute the style and content losses at the pre-selected convolutional layers. We add style losses to get the total style loss and content losses to get the total content loss. Finally, we obtain a total loss by summing these two components in a weighted fashion.

If we give more weight to the style component, the generated image will have more style reflected on it and vice versa. Using gradient descent, we backpropagate the loss all the way back to the input in order to update our generated image. After a few epochs, the generated image should evolve in a way that it produces the content and style representations that minimize the respective losses, thereby producing a style transferred image.

In the preceding diagram, the pooling layer is average pooling-based instead of the traditional max pooling. Average pooling is deliberately used for style transfer to ensure smooth gradient flow. We want the generated images not to have sharp changes between pixels. Also, it is worth noticing that the network in the preceding diagram ends at the layer where the last style or content loss is calculated. Hence, in this case, because there is no loss associated with the sixth convolutional layer of the original network, it is meaningless to talk about layers beyond the fifth convolutional layer in the context of style transfer.

In the next section, we will implement our own neural style transfer system using PyTorch. With the help of a pre-trained VGG model, we will use the concepts we've discussed in this section to generate artistically styled images. We will also explore the impact of tuning the various model parameters on the content and texture/style of generated images.

Implementing neural style transfer using PyTorch

Having discussed the internals of a neural style transfer system, we are all set to build one using PyTorch. In the form of an exercise, we will load a style and a content image. Then, we will load the pre-trained VGG model. After defining which layers to compute the style and content loss on, we will trim the model so that it only retains the relevant layers. Finally, we will train the neural style transfer model in order to refine the generated image epoch by epoch.

Loading the content and style images

In this exercise, we will only show the important parts of the code for demonstration purposes. To access the full code, go to our github repository [8.3] . Follow these steps:

1. Firstly, we need to import the necessary libraries :

```
from PIL import Image
import matplotlib.pyplot as pltimport torch
import torch.nn as nn
import torch.optim as optim
import torchvisiondvc = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Amng other libraries, we import the `torchvision` library to load the pre-trained VGG model and other computer vision-related utilities.

1. Next, we need a style and a content image. We will use the unsplash website [8.4] to download an image of each kind. The downloaded images are included in the code repository of this book. In the following code, we are writing a function that will load the images as tensors:

```
def image_to_tensor(image_filepath,  image_dimension=128):
    img = Image.open(image_filepath).convert('RGB')
    # display image
    …
    torch_transformation =       torchvision.transforms.Compose([
        torchvision.transforms.Resize(img_size),
        torchvision.transforms.ToTensor()
                                ])
    img = torch_transformation(img).unsqueeze(0)
    return img.to(dvc, torch.float)
style_image = image_to_tensor("./images/style.jpg")
content_image =image_to_tensor("./images/content.jpg")
```

This should give us the following output:

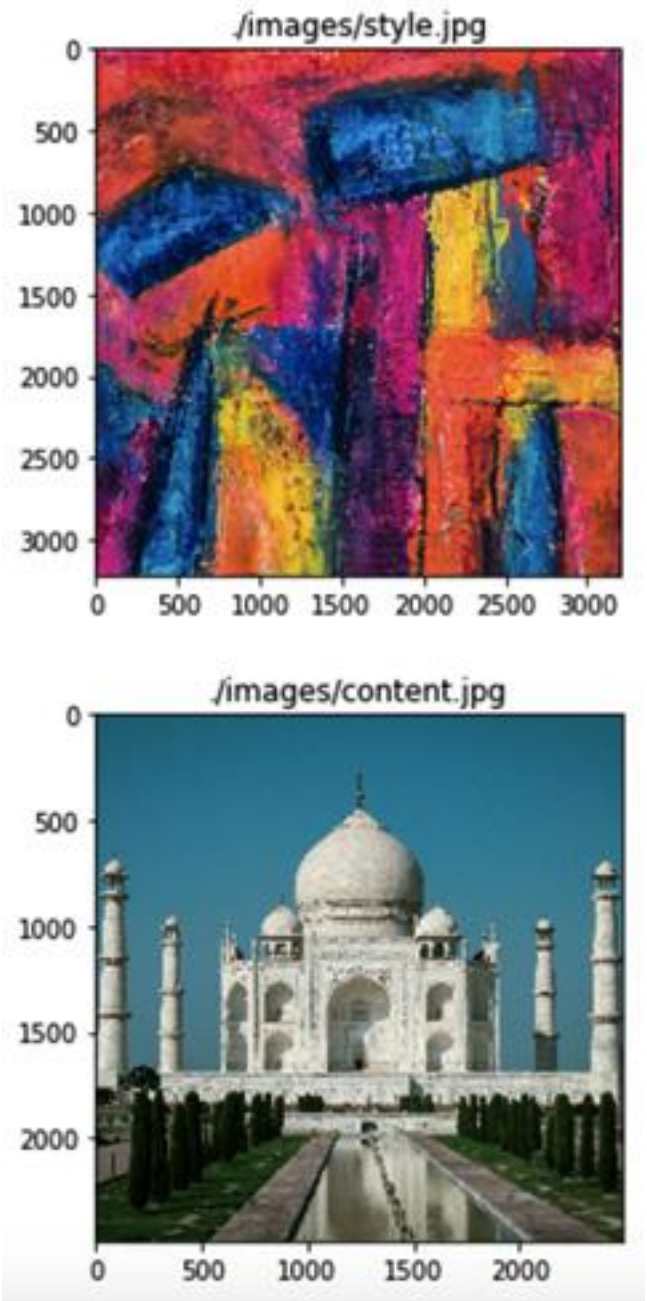

Figure 8. 5 – Style and content images

So, the content image is a real-life photograph of the *Taj Mahal*, whereas the style image is an art painting. Using style transfer, we hope to generate an artistic *Taj Mahal* painting. However, before we do that, we need to load and trim the VGG19 model.

Loading and trimming the pre-trained VGG19 model

In this part of the exercise, we will use a pre-trained VGG model and retain its convolutional layers. We will make some minor changes to the model to make it usable for neural style transfer. Let's get started:

1. We will first load the pre-trained VGG19 model and use its convolutional layers to generate the content and style targets to yield the content and style losses, respectively:

```
vgg19_model = torchvision.models.vgg19(pretrained=True).to(dvc)
print(vgg19_model)
```

The output should be as follows:

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (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=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (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=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (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=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace=True)
    (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=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Figure 8. 6 – VGG19 model

1. We do not need the linear layers; that is, we only need the convolutional part of the model. In the preceding code, this can be achieved by only retaining the `features` attribute of the model object, as follows:

```
vgg19_model = vgg19_model.features
```

Note

In this exercise, we are not going to tune the parameters of the VGG model. All we are going to tune is the pixels of the generated image, right at the input end of the model. Hence, we will ensure that the parameters of the loaded VGG model are fixed.

1. We must freeze the parameters of the VGG model with the following code:

```
for param in vgg19_model.parameters():
    param.requires_grad_(False)
```

1. Now that we've loaded the relevant section of the VGG model, we need to change the `maxpool` layers into average pooling layers, as discussed in the previous section. While doing so, we will take note of where the convolutional layers are located in the model:

```
conv_indices = []for i in range(len(vgg19_model)):
    if vgg19_model[i]._get_name() == 'MaxPool2d':
        vgg19_model[i] =  nn.AvgPool2d(kernel_size=vgg19_model[i].kernel_size,
stride=vgg19_model[i].stride, padding=vgg19_model[i].padding)
    if vgg19_model[i]._get_name() == 'Conv2d':
        conv_indices.append(i)
conv_indices = dict(enumerate(conv_indices, 1))print(vgg19_model)
```

The output should be as follows:

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace=True)
  (9): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU(inplace=True)
  (18): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (24): ReLU(inplace=True)
  (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (26): ReLU(inplace=True)
  (27): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace=True)
  (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (31): ReLU(inplace=True)
  (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (33): ReLU(inplace=True)
  (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (35): ReLU(inplace=True)
  (36): AvgPool2d(kernel_size=2, stride=2, padding=0)
)
```

Figure 8. 7 – Modified VGG19 model

As we can see, the linear layers have been removed and the max pooling layers have been replaced by average pooling layers, as indicated by the red boxes in the preceding figure.

In the preceding steps, we loaded a pre-trained VGG model and modified it in order to use it as a neural style transfer model. Next, we will transform this modified VGG model into a neural style transfer model.

Building the neural style transfer model

At this point, we can define which convolutional layers we want the content and style losses to be calculated on. In the original paper, style loss was calculated on the first five convolutional layers, while content loss was calculated on the fourth convolutional layer only. We will follow the same convention, although you are welcome to try out different combinations and observe their effects on the generated image. Follow these steps:

1. First, we list the layers we need to have the style and content loss on:

```
layers = {1: 's', 2: 's', 3: 's', 4: 'sc', 5: 's'}
```

Here, we have defined the first to fifth convolutional layers, which are attached to the style loss, and the fourth convolutional layer, which is attached to the content loss.

1. Now, let's remove the unnecessary parts of the VGG model. We shall only retain it until the fifth convolutional layer, as shown here:

```
vgg_layers = nn.ModuleList(vgg19_model)
last_layer_idx = conv_indices[max(layers.keys())]
vgg_layers_trimmed = vgg_layers[:last_layer_idx+1]
neural_style_transfer_model = nn.Sequential(*vgg_layers_trimmed)
print(neural_style_transfer_model)
```

This should give us the following output:

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU()
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU()
  (4): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU()
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU()
  (9): AvgPool2d(kernel_size=2, stride=2, padding=0)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
)
```

Figure 8. 8 – Neural style transfer model object

As we can see, we have transformed the VGG model with 16 convolutional layers into a neural style transfer model with five convolutional layers.

Training the style transfer model

In this section, we'll start working on the image that will be generated. We can initialize this image in many ways, such as by using a random noise image or using the content image as the initial image. Currently, we are going to start with random noise. Later, we will also see how using the content image as the starting point impacts the results. Follow these steps:

1. The following code demonstrates the process of initializing a `torch` tensor with random numbers:

```
# initialize as the content image
# ip_image = content_image.clone()
# initialize as random noise:
ip_image = torch.randn(content_image.data.size(), device=dvc)
plt.figure()
plt.imshow(ip_image.squeeze(0).cpu().detach().numpy().transpose(1,2,0).clip(0,1));
```

This should give us the following output:

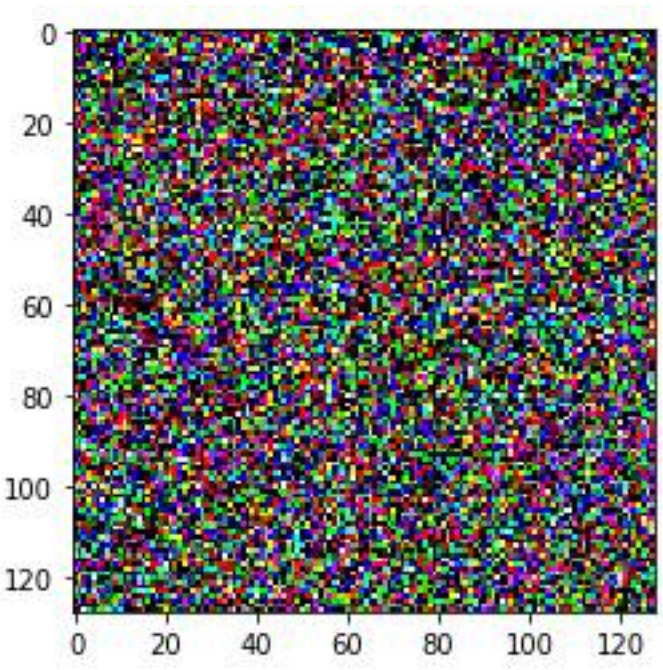

Figure 8. 9 – Random noise image

1. Finally, we can start the model training loop. First, we will define the number of epochs to train for, the relative weightage to provide for the style and content losses, and instantiate the Adam optimizer for gradient descent-based optimization with a learning rate of `0.1`:

```
num_epochs=180
wt_style=1e6
wt_content=1
style_losses = []
content_losses = []
opt = optim.Adam([ip_image.requires_grad_()], lr=0.1)
```

1. Upon starting the training loop, we initialize the style and content losses to zero at the beginning of the epoch, and then clip the pixel values of the input image between `0` and `1` for numerical stability:

```
for curr_epoch in range(1, num_epochs+1):
    ip_image.data.clamp_(0, 1)
    opt.zero_grad()
    epoch_style_loss = 0
    epoch_content_loss = 0
```

1. At this stage, we have reached a crucial step in the training iteration. Here, we must calculate the style and content losses for each of the pre-defined style and content convolutional layers. The individual style losses and content losses for each of the respective layers are added together to get the total style and content loss for the current epoch:

```
    for k in layers.keys():
        if 'c' in layers[k]:
            target = neural_style_transfer_model[:conv_indices[k]+1]
(content_image).detach()
            ip = neural_style_transfer_model[:conv_indices[k]+1](ip_image)
            epoch_content_loss += torch.nn.functional.mse_loss(ip, target)
        if 's' in layers[k]:
            target = gram_matrix(neural_style_transfer_model[:conv_indices[k]+1]
(style_image)).detach()
            ip = gram_matrix(neural_style_transfer_model[:conv_indices[k]+1](ip_image))
            epoch_style_loss += torch.nn.functional.mse_loss(ip, target)
```

As shown in the preceding code, for both the style and content losses, first, we compute the style and content targets (ground truths) using the style and content image. We use `.detach()` for the targets to indicate that these are not trainable but just fixed target values. Next, we compute the predicted style and content outputs based on the generated image as input, at each of the style and content layers. Finally, we compute the style and content losses.

1. For the style loss, we also need to compute the gram matrix using a pre-defined gram matrix function, as shown in the following code:

```
def gram_matrix(ip):
    num_batch, num_channels, height, width = ip.size()
    feats = ip.view(num_batch * num_channels, width *   height)
    gram_mat = torch.mm(feats, feats.t())
    return gram_mat.div(num_batch * num_channels *        width * height)
```

As we mentioned earlier, we can compute an inner dot product using the `torch.mm` function. This computes the gram matrix and normalizes the matrix by dividing it by the number of feature maps times the width times the height of each feature map.

1. Moving on in our training loop, now that we've computed the total style and content losses, we need to compute the final total loss as a weighted sum of these two, using the weights we defined earlier:

```
    epoch_style_loss *= wt_style
    epoch_content_loss *= wt_content
    total_loss = epoch_style_loss + epoch_content_loss
    total_loss.backward()
```

Finally, at every *k* epochs, we can see the progression of our training by looking at the losses as well as looking at the generated image. The following figure shows the evolution of the generated style transferred image for the previous code for a total of 180 epochs recorded at every 20 epochs:

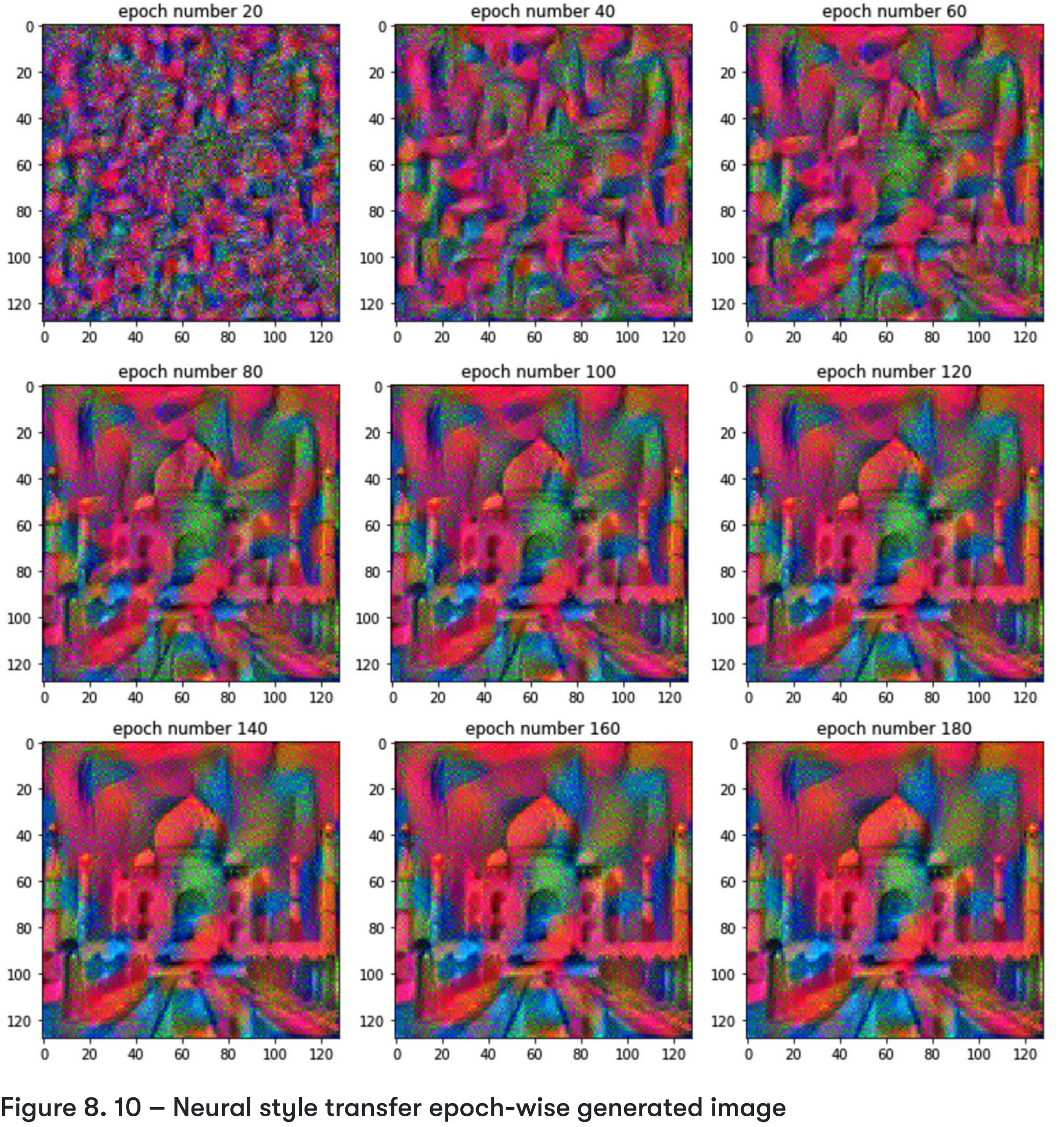

Figure 8. 10 – Neural style transfer epoch-wise generated image

It is quite clear that the model begins by applying the style from the style image to the random noise. As training proceeds, the content loss starts playing its role, thereby imparting content to the styled image. By epoch **180**, we can see the generated image, which looks like a good approximation of an artistic painting of the *Taj Mahal*. The following graph shows the decreasing style and content losses as the epochs progress from **0** to **180**:

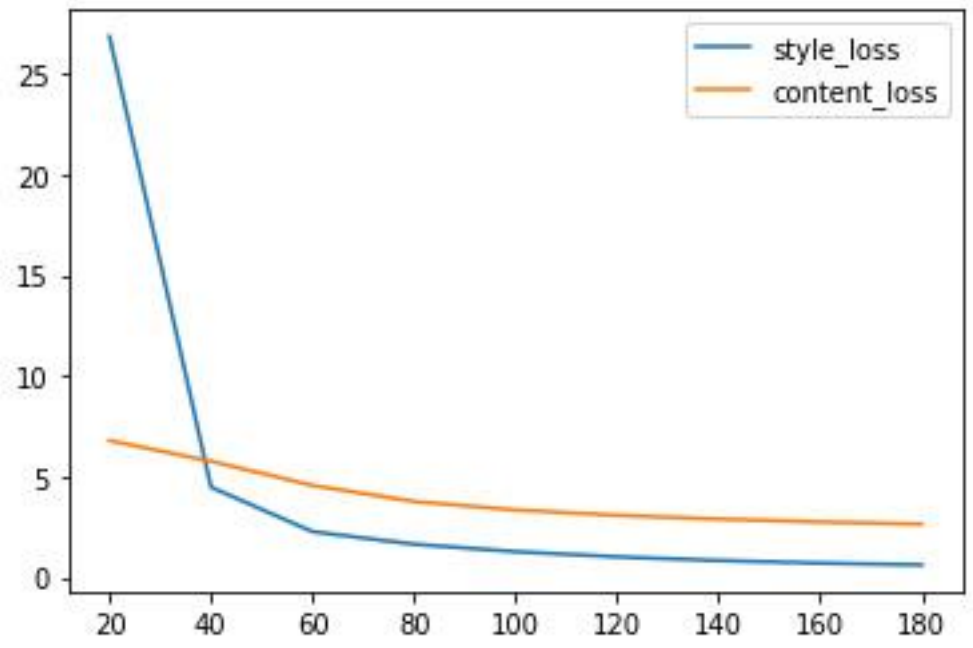

Figure 8. 11 – Style and content loss curves

Noticeably, the style loss sharply goes down initially, which is also evident in *Figure 8. 10* in that the initial epochs mark the imposition of style on the image more than the content. At the advanced stages of training, both losses decline together gradually, resulting in a style transferred image, which is a decent compromise between the artwork of the style image and the realism of a photograph that's been taken with a camera.

Experimenting with the style transfer system

Having successfully trained a style transfer system in the previous section, we will now look at how the system responds to different hyperparameter settings. Follow these steps:

1. In the preceding section, we set the content weight to `1` and the style weight to `1e6`. Let's increase the style weight 10x further – that is, to `1e7` – and observe how it affects the style transfer process. Upon training with the new weights for 600 epochs, we get the following progression of style transfer:

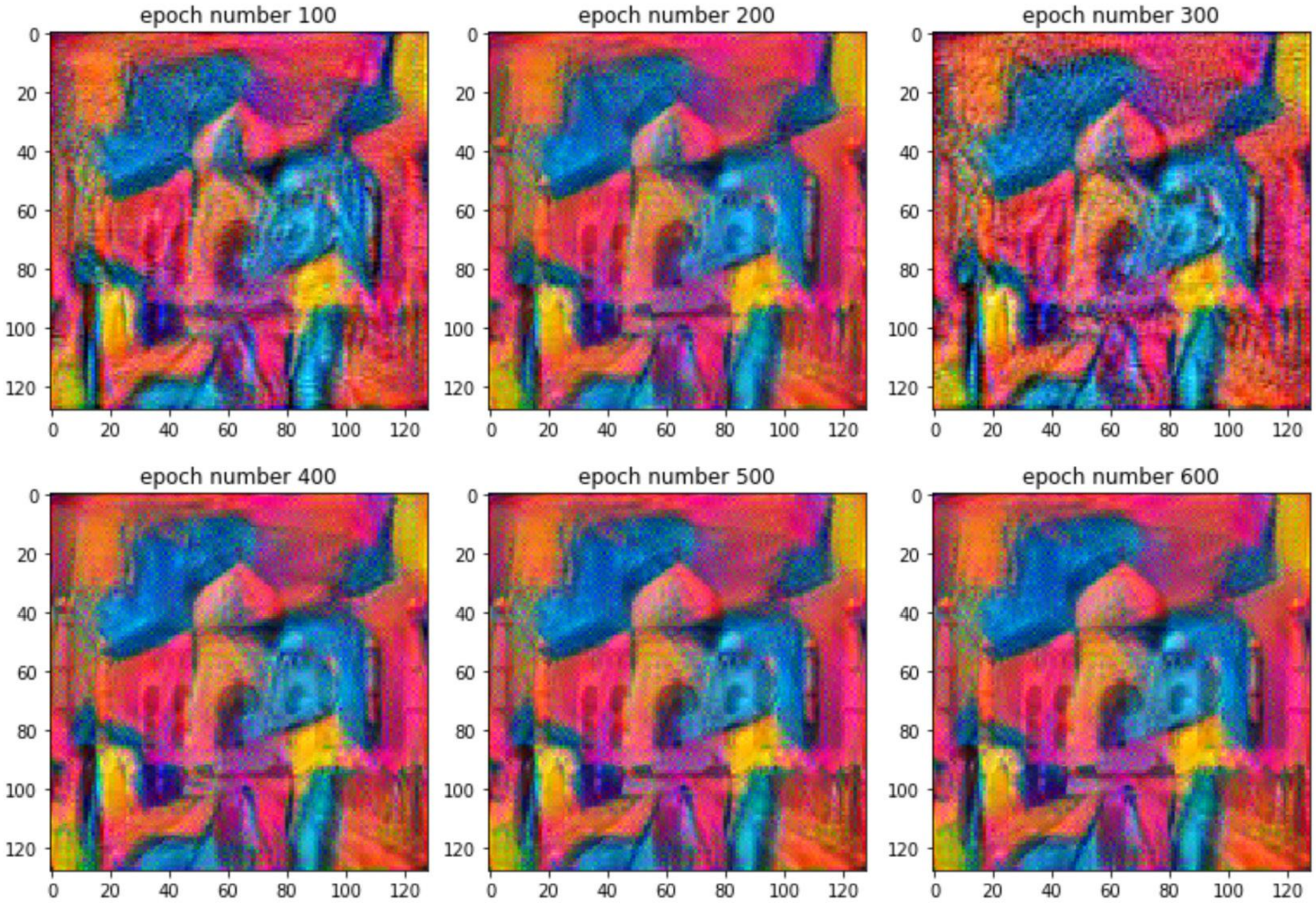

Figure 8. 12 – Style transfer epochs with higher style weights

Here, we can see that initially, it required many more epochs than in the previous scenario to reach a reasonable result. More importantly, the higher style weight does seem to have an effect on the generated image. When we look at the images in the preceding figure compared to the ones in *Figure 8. 10*, we find that the former have a stronger resemblance to the style image shown in *Figure 8. 5*.

1. Likewise, reducing the style weight from `1e6` to `1e5` produces a more content-focused result, as can be seen in the following screenshot:

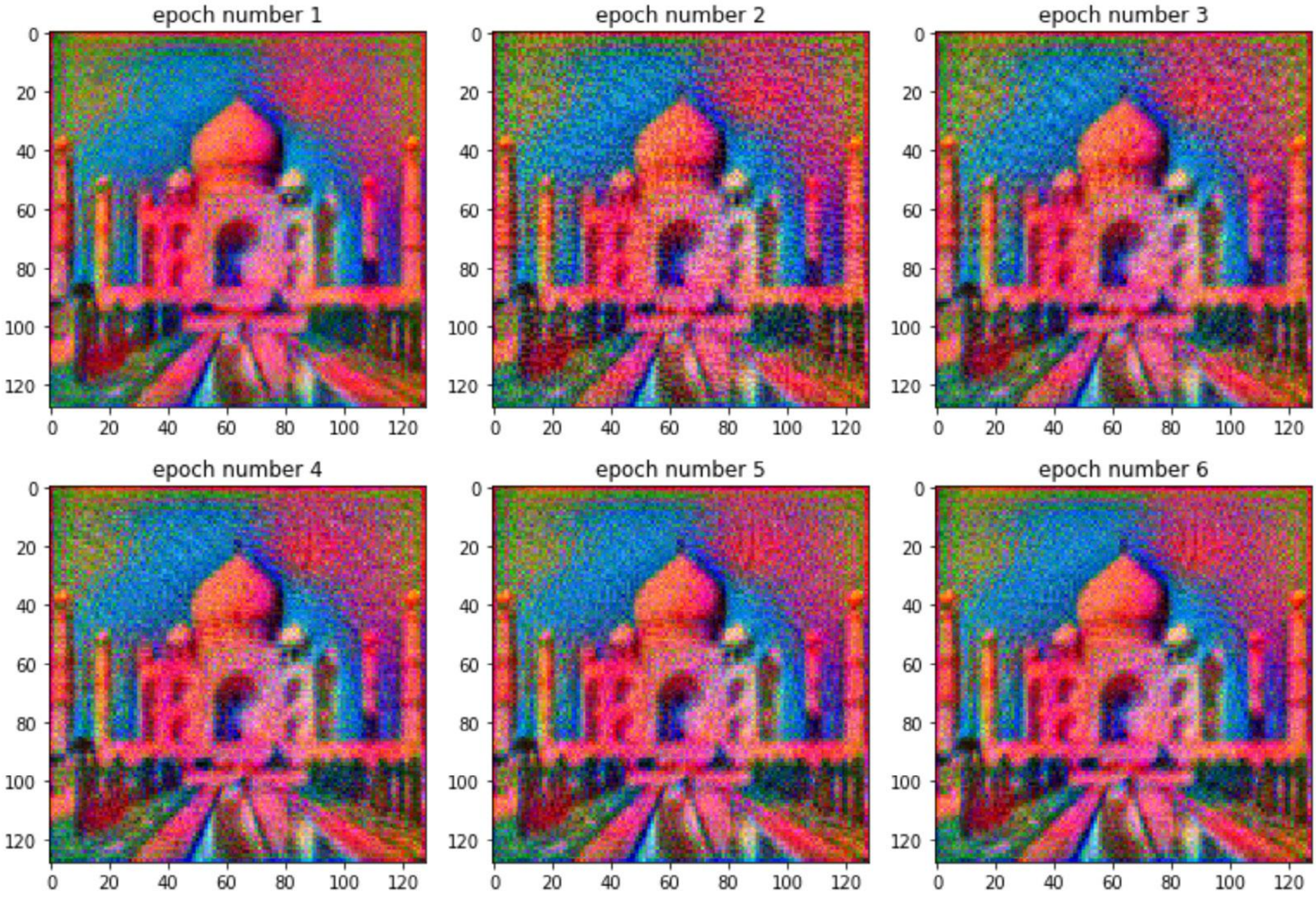

Figure 8. 13 – Style transfer epochs with lower style weights

Compared to the scenario with a higher style weight, having a lower style weight means it takes far fewer epochs to get a reasonable-looking result. The amount of style in the generated image is much smaller and is mostly filled with the content image data. We only trained this scenario for 6 epochs as the results saturate after that point.

1. A final change could be to initialize the generated image with the content image instead of the random noise, while using the original style and content weights of `1e6` and `1`, respectively. The following figure shows the epoch-wise progression in this scenario:

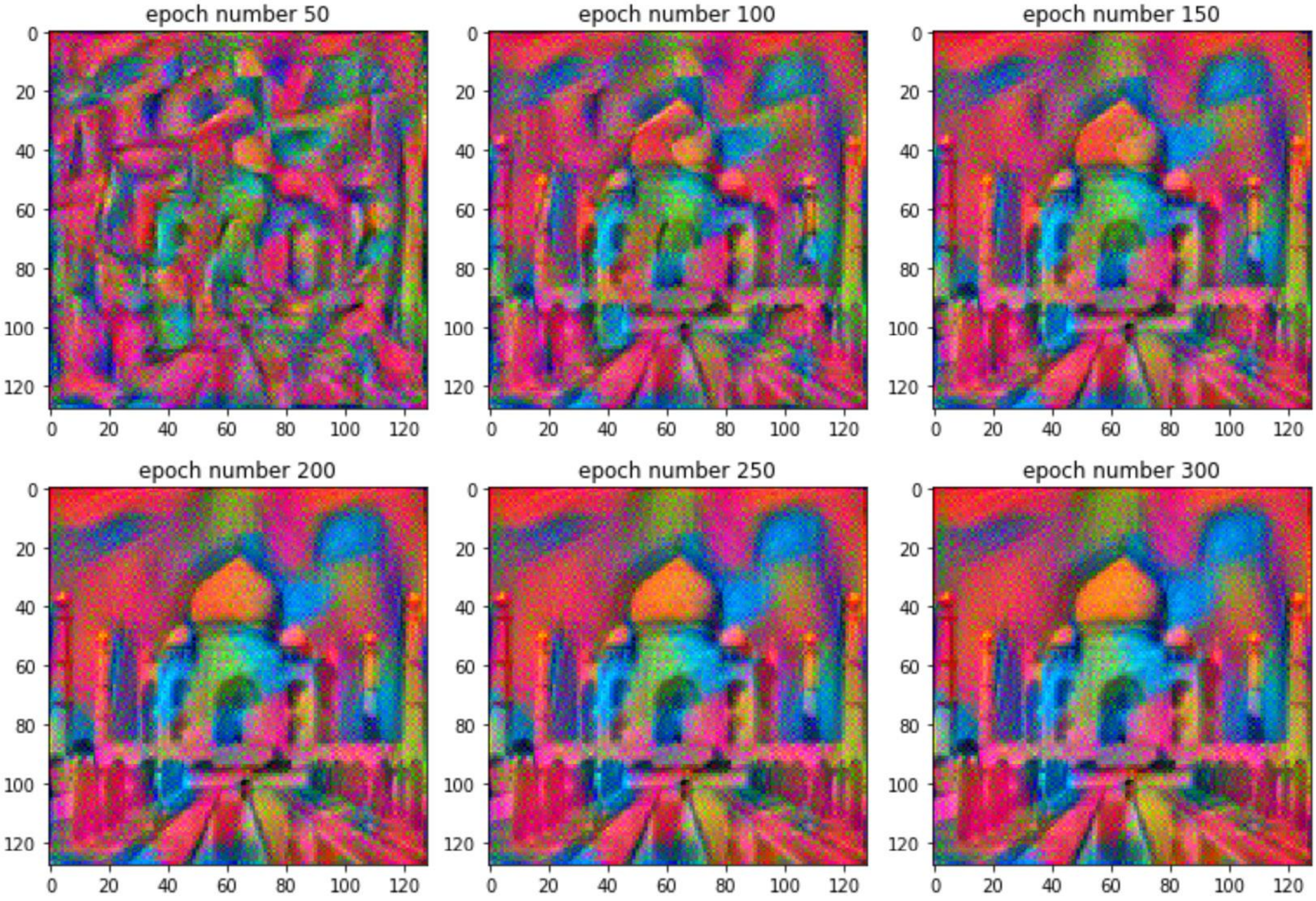

Figure 8. 14 – Style transfer epochs with content image initialization

By comparing the preceding figure to *Figure 8. 10*, we can see that having the content image as a starting point gives us a different path of progression to getting a reasonable style transferred image. It seems that both the content and style components are being imposed on the generated image more simultaneously than in *Figure 8. 10*, where the style got imposed first, followed by the content. The following graph confirms this hypothesis:

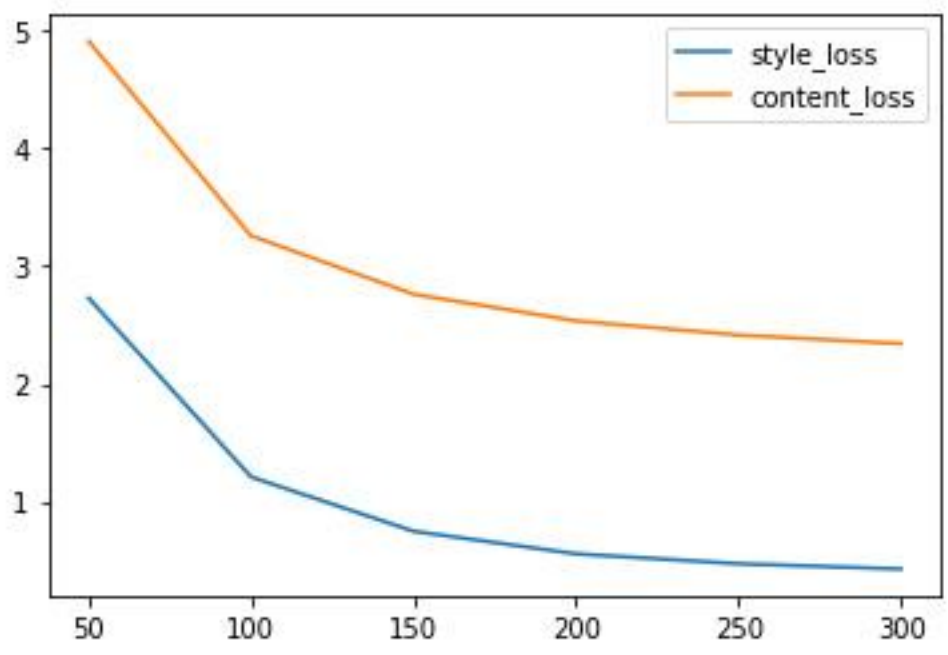

Figure 8. 15 – Style and content loss curves with content image initialization

As we can see, both style and content losses are decreasing together as the epochs progress, eventually saturating toward the end. Nonetheless, the end results in both *Figures 8. 10* and 8. *14* or even *Figures 8. 12* and 8. *13* all represent reasonable artistic

impressions of the *Taj Mahal*.

We have successfully built a neural style transfer model using PyTorch, wherein using a content image – a photograph of the beautiful *Taj Mahal* – and a style image – a canvas painting – we generated a reasonable approximation of an artistic painting of the *Taj Mahal*. This application can be extended to various other combinations. Swapping the content and style images could also produce interesting results and give more insight into the inner workings of the model.

You are encouraged to extend the exercise we discussed in this chapter by doing the following:

- Changing the list of style and content layers
- Using larger image sizes
- Trying more combinations of style and content loss weights
- Using other optimizers, such as SGD and LBFGS
- Training for longer epochs with different learning rates, in order to observe the differences in the generated images across all these approaches

Summary

In this chapter, we applied the concept of generative machine learning to images by generating an image that contains the content of one image and the style of another – a task known as neural style transfer. In the next chapter, we will expand on this paradigm, where we'll have a generator that generates *fake* data and there is a discriminator that tells apart *fake* data from *real* data. Such models are popularly known as **generative adversarial networks (GANs)**. We will be exploring deep convolutional GANs (DCGANs) in the next chapter.

Previous Chapter

Next Chapter