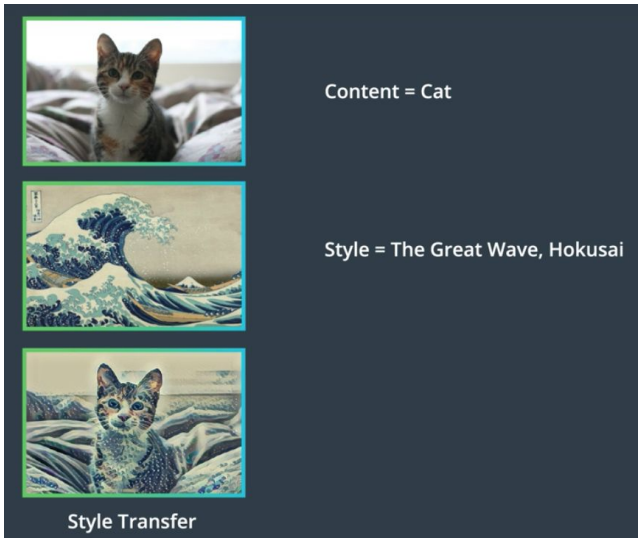
Lesson 7: Style Transfer

Notes by: Jvinniec

Introduction

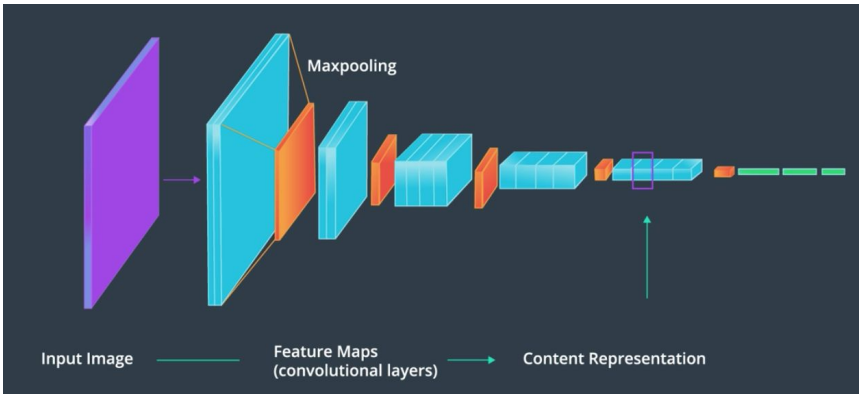
Style transfer takes the information learned from one image and applies it to another image:



This is done by using a trained CNN to separate the style of an image from the content of the image.

Separating Style and Content

As we train a CNN deeper and deeper we increasingly train a network that cares more and more about the content of the images than the underlying style. Later layers are even sometimes referred to as a "**content representation**".



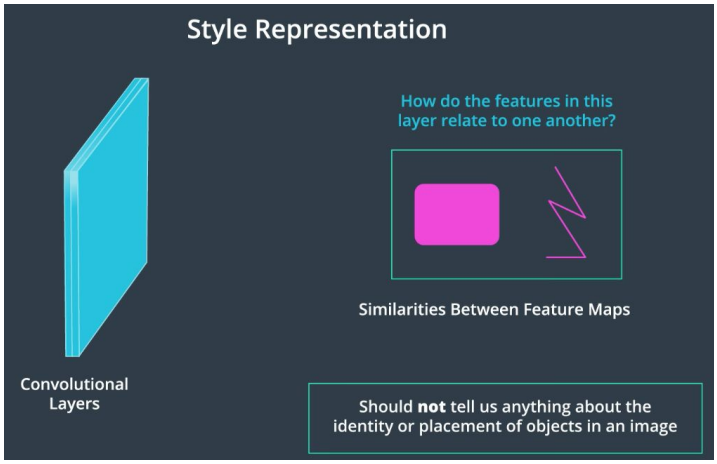
Style is analogous to the traits that might be found in the brush strokes of a painting:

- **Texture** (e.g. what type of paints were used)
- **Colors** (bright, dark, softer, neon)
- **Curvature** (are they straight or curved strokes)

Similarities Between Feature Maps

To capture only the style portion, a feature space designed to capture texture and color is used. This feature space looks at spatial correlations within a layer of a network. Basically, we look at the generated convolutional layers of a map and see which features are strongly correlated

between the generated kernels (e.g. which colors are most strongly detected, are sharp edges or curved edges more frequently detected, etc…).



Paper Overview

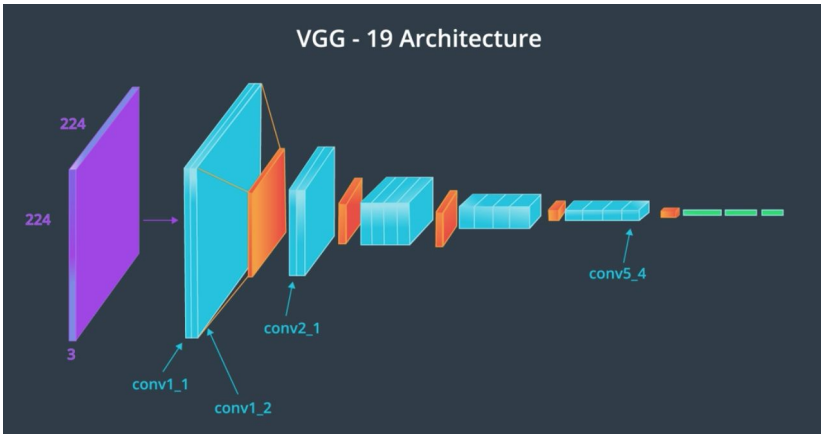
An interesting starting point might be the paper "Image Style Transfer Using Convolutional Neural Networks".

Style Transfer Algorithm

Here we will discuss a few examples of style transfer algorithms.

VGG19

VGG19 is a CNN trained with 19 layers. In between the maxpooling layers (orange) are stacks of 2 or 4 convolutional layers (blue).



Style Transfer Parameters

The following are some important measures when applying style transfer.

Content Loss

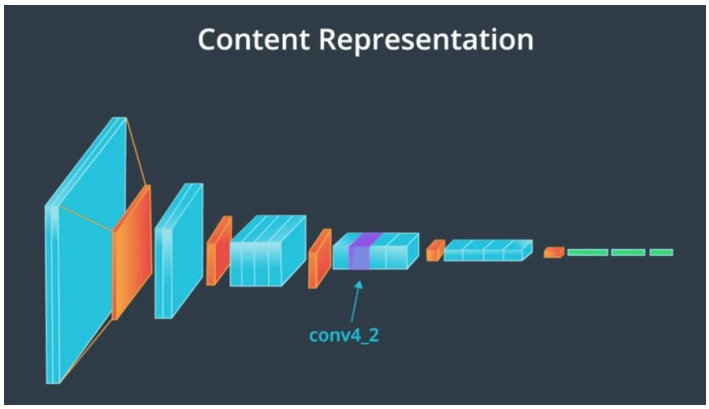
Content Loss is a measure of how much the content of our image has changed as we apply our style transfer algorithm. Ideally we want to minimize our content loss. To compute content loss we need to compare two different representations:

- **Content Image (C_c)**: The baseline input image.
- **Target Image (T_c)**: The output from the style transfer algorithm.

Ideally, the content in both images will be approximately equivalent. The content loss is represented by:

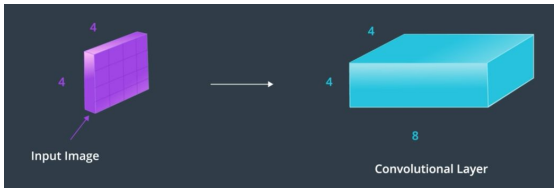
$$L_{content} = \frac{1}{2} \sum (T_c - C_c)^2$$

In the VGG19 algorithm, they use fourth convolutional layer for their output target image.



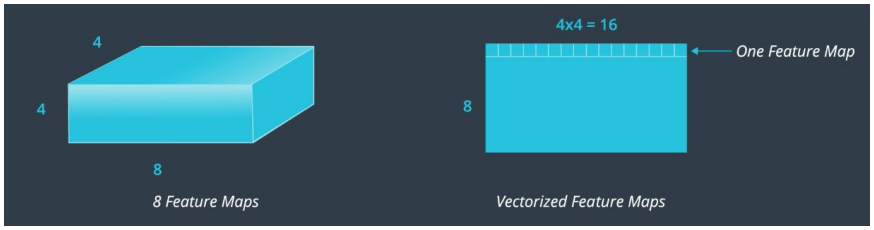
Gram Matrix

The Gram matrix represents the correlations between the given feature maps in a convolutional layer. Consider a simple 4x4 image passed through a convolutional layer that has 8 feature maps. The resulting output will be 4x4x8 (height x width x depth).

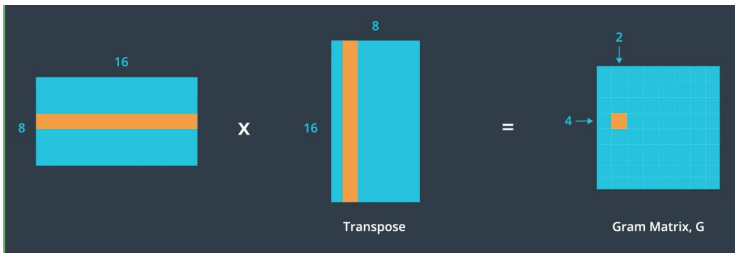


This output now has 8 feature maps that we want to find the correlations between. To compute the Gram matrix:

1. **Flatten the individual feature maps** (a.k.a. Generate a vectorized feature map). The resulting output from the above example will have dimensions of 16 x 8.



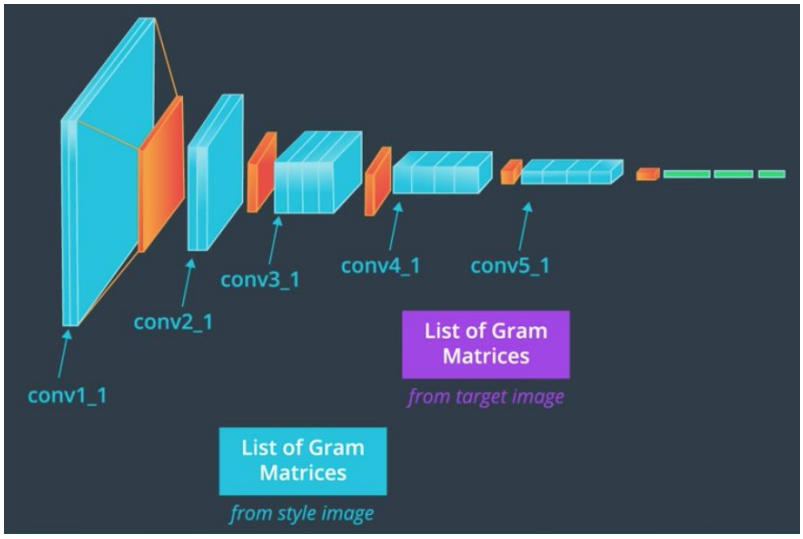
2. **Multiply the output matrix by its transpose.** The resulting image is 8 x 8. This treats each value in a feature map as an individual sample unrelated spatially to other values. Thus the Gram matrix contains non-localized information about the layer (i.e. info that would still be there even if the image was shuffled).



Style Loss

Just as it is important to track how much we've changed from the content of our input image, we also need to track how much we've changed from the style of our style image.

To compute the style loss, we compare the Gram matrices at each convolutional layer:



The style loss is given by:

$$L_{style} = a \sum_i w_i (T_{si} - S_{si})^2$$

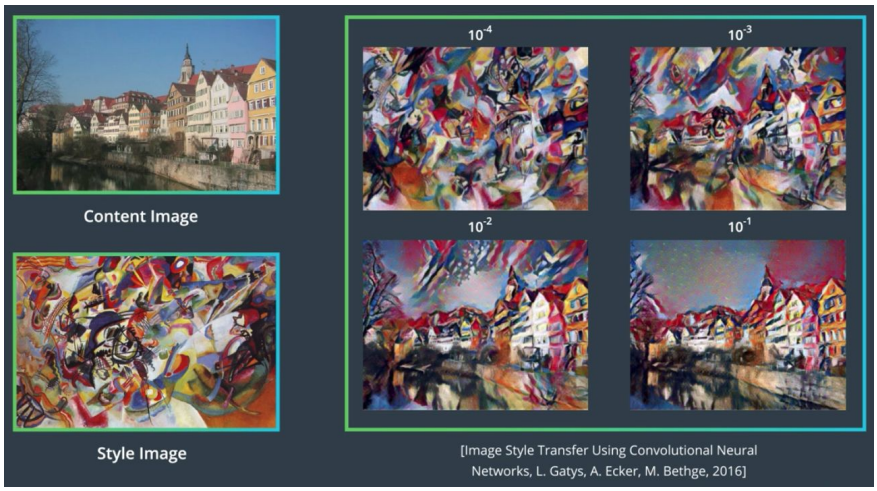
where *i* iterates over the convX_i Gram matrices. T_{si} is the *i*-th Gram matrix for our target image and S_{si} is the Gram matrix of our input style image. The parameters *a* and w_i are allow us to scale the loss formula to increase or decrease the style loss.

Combining Loss Terms

Together the content loss and style loss allow us to minimize the loss of the content in our input image while also minimizing the loss of our desired style from our input style image.

We want to consider both loss terms equally, so we typically need to scale the values by some constants:

Note that as the ratio of alpha:beta decreases (i.e. larger relative beta) we end up with more influence from the style image and a greater loss in our content:



Style Transfer in PyTorch

Applying the style transfer algorithm in PyTorch involves following steps (more details later on):

1. **Load images and model**
2. **Identify features layers** in the model that will serve as features for the content and style loss.
    a. Content loss uses a single layer from the network
    b. Style loss is a weighted average of several layers in the network.
3. **Setup the optimizer** to use the values of the pixels in our output image as the parameters to be optimized (initialize them to the values in the input content image)
4. **Setup the weights**:

    a. For the relative weight of content and style losses
    b. For weighting the individual feature layers used to compute style loss
5. **Iterate on the image** until we have a nice looking output image
    a. Involves repeatedly computing the Gram matrices

Load Images and Model

First we load the content and style images, ensuring that they are the same x,y,z size. This is necessary for a pixel-by-pixel comparison in the loss calculations. Also, if using a GPU make sure to transfer the images to the appropriate device.

The model used in the course example is the VGG19 model and can be downloaded in the following way:

```
# 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)
```

Note that we do not need to compute the gradients of the parameters in the matrix as we will not be updating them.

Identify The Feature Layers

In this step we need to identify which layers of the model we will use for computing the content and style loss. For the example in the course we will use:

- Convolutional layers 0 (‘`conv1_1`’), 5 (‘`conv2_1`’), 10 (‘`conv3_1`’), 19 (‘`conv4_1`’), and 28 (‘`conv5_1`’) for computing the style loss
- Convolutional layer 21 (‘`conv4_2`’) for computing the content loss

Setup the Optimizer

The optimizer now uses the values of the pixels in the input image as the variables to be optimized. First we create a copy of the content (input) image, making sure to specify that the values in the cloned tensor require gradient computation and that we move it to the appropriate device:

```
target = content.clone().requires_grad_(True).to(device)
```

Then we pass this object to the optimizer for optimization:

```
optimizer = optim.Adam([target], lr=0.003)
```

Setup the Weights

Relative Content and Style Loss Weight

The content and style loss should be weighted to prioritize content vs. style appropriately. Typically this is done by setting the content loss weight to 1 and setting the style loss weight to a very large number (like 1e6) and adjusting only the style weight as needed:

```
content_weight = 1     # alpha
style_weight   = 1e6   # beta
```

Note that a larger style weighing means a greater priority is given to the style than to the content.

Relative Style Feature Weighting

We also need to set the weights for the feature layers used to compute the style loss. For this we can setup a Python 'dict' with the layer names and the associated weights:

```
style_weights = {'conv1_1': 1.,
                 'conv2_1': 0.75,
                 'conv3_1': 0.1,
                 'conv4_1': 0.1,
                 'conv5_1': 0.1}
```

Note here that the earlier layers (like '`conv1_1`') typically represent larger features and the later layers represent smaller, more fine features in the image. Setting the weights appropriately will control whether larger or finer features are preferred.

Iterate on the Image

Now that we have all of the weights established and our optimizer setup, we can iterate on the image just like we would before, just with a more complex computation of the loss.

First we need to compute the resulting feature images of the target image:

```
target_features = get_features(image=target, model=vgg)
```

Next we get the content loss:

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

```
                                content_features['conv4_2'])**2)
```

Next we loop on the style feature layers to compute 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]
    _, d, h, w = target_feature.shape

    # Calculate the target gram matrix
    target_gram = gram_matrix(target_feature)

    # Get the "style" style representation
    style_gram = style_grams[layer]

    # Calculate the style loss for one layer, weighted appropriately
    layer_style_loss = style_weights[layer] *
                       torch.mean((target_gram - style_gram)**2)

    # Add to the style loss
    style_loss += layer_style_loss / (d * h * w)
```

Note that the above involves computing the Gram matrix for the target and style feature images, which is done in the following way (note that '`image`' is represented as a PyTorch tensor):

```
# Get the batch_size, depth, height, and width of the Tensor
batch_size, d, h, w = image.size()
# Reshape it, so we're multiplying the features for each channel
matrix = image.view(d, h*w)
# Multiply the matrix by its transpose to get the Gram Matrix
gram = torch.mm(matrix, matrix.t())
```

Now combine the loss terms:

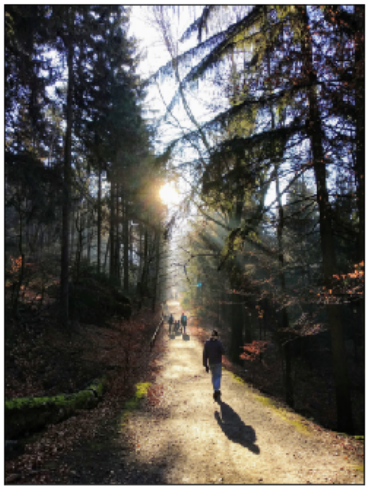
```
# Calculate the *total* loss
total_loss = content_weight*content_loss + style_weight*style_loss
```

Now we can zero the gradients in the optimizer, run our back propagation with the 'total_loss' parameter, and step the optimizer, just like we've done before.

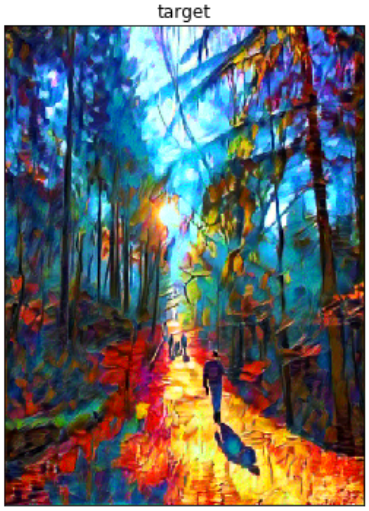
A Worked Example

Here's a worked example I generated from the above algorithm:

content



target



style