Implementing style transfer in images

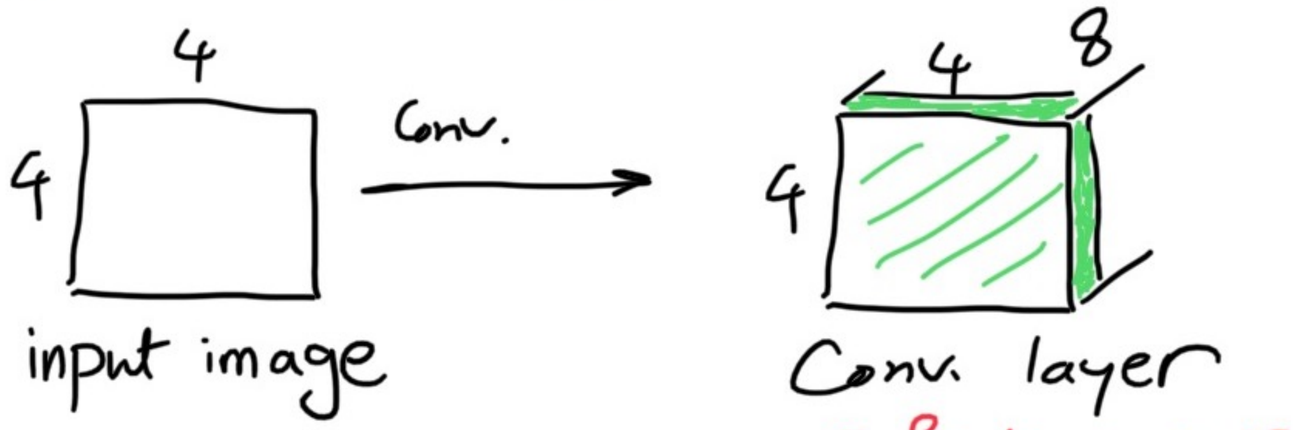
Content Image → VGG19 Arch. → the output of the deepest Conv. layer in vgg19 is the content representation.

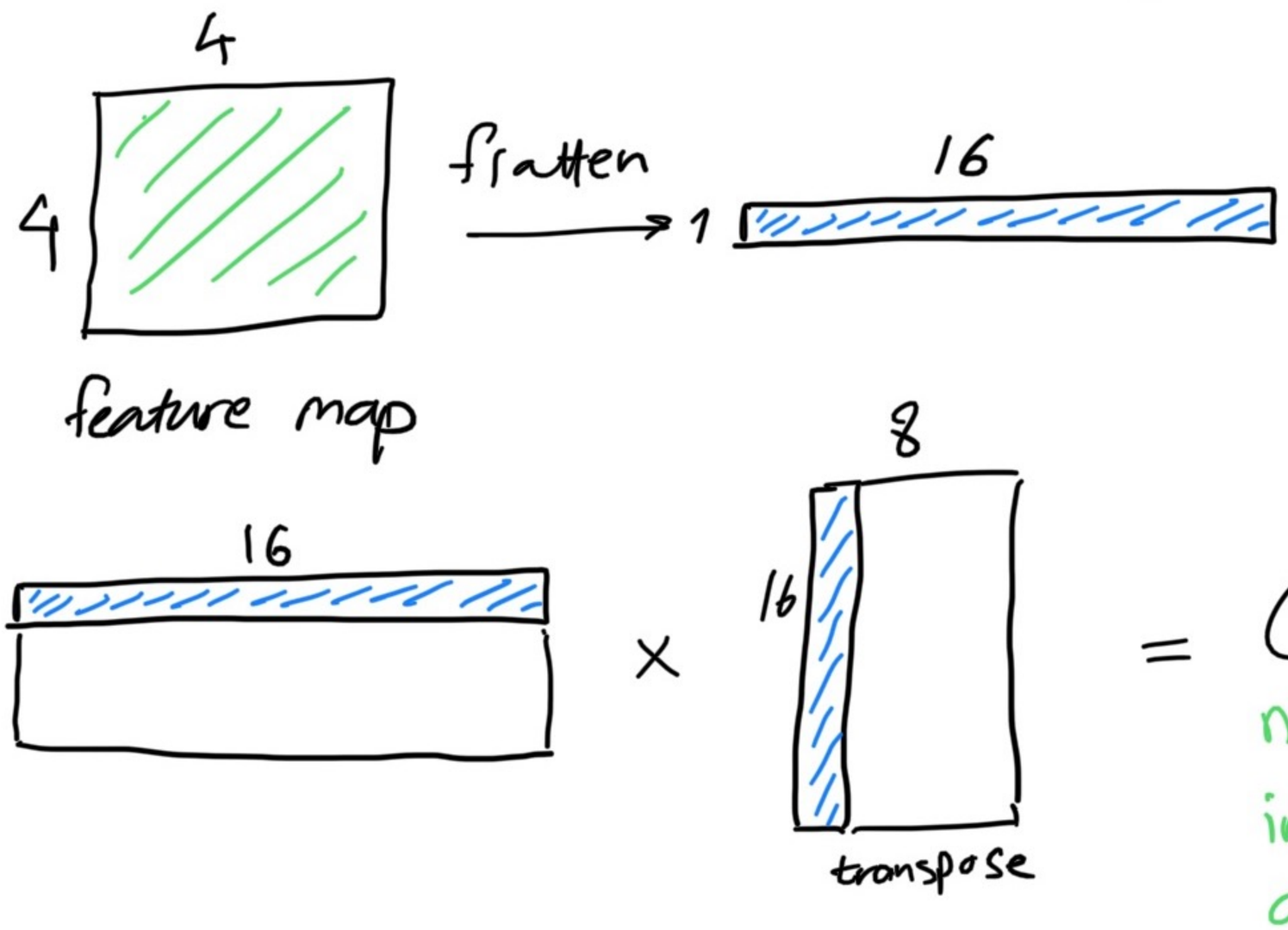
Style Image → VGG19 Arch. → we get outputs of multiple Conv. layers

Content representation (content image) C_c ≈ (ideally) Content representation (target image) T_c

$$\underbrace{\mathcal{L}_{Content}}_{\text{Content loss}} = \frac{1}{2} \sum (T_c - C_c)^2$$

Gram Matrix





List of Gram Matrices of style image (S_s)
List of Gam Matrices of target image (T_s)

$$\underbrace{\mathcal{L}_{style}}_{\text{Style loss}} = a \sum_i w_i \left(T_{s,i} - S_{s,i} \right)^2$$

Total loss = α $\mathcal{L}_{content}$ + β $\mathcal{L}_{style}$

α: content weight
β: style weight

- Style weight is usually much larger than the Content weight.
- to add more style to target image, decrease $\frac{\alpha}{\beta}$.

In PyTorch:

```
from PIL import Image → load any image
from io import BytesIO
import numpy as np

import torch
import torch.optim as optim
import requests
from torchvision import transforms, models
                                    Conv. and pooling layers
# get feature layers of vgg19:      ↑
vgg = models.vgg19(pretrained = True).features

# freeze the parameters:
for param in vgg.parameters():
    param.requires_grad_(False)
```

```
# move to device
device = torch.device("cuda" if torch.cuda.is_available()
                      else "cpu")
vgg.to(device)

# Load Content/style images
def load_image(img_path, max_size=400,
               shape=None):
    if "http" in img_path:
        response = requests.get(img_path)
        image = Image.open(BytesIO(response.content))
                      .Convert('RGB')
    else:
        image = Image.open(img_path).Convert('RGB')
```

actual loading

resize all to a managable size

```
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 alpha channel →

```
image = in_transform(image)[:3,:,:].unsqueeze(0)
return image
```

```
content = load_image('img1.jpg').to(device)
style = load_image('img2.jpg', shape=content.shape[-2:]).
                                                to(device)
```

to align easier

un-normalize an image

```
def im_Convert(tensor):
    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
```

```
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(20,10))
ax1.imshow(im_Convert(content))
ax2.imshow(im_Convert(style))
```

pass image through layers

```
def get_features(image, model, layers=None):
    if layers is None:
        layers = {'0': 'Conv1-1',
                  '5': 'Conv2-1',
                  '10': 'Conv3-1',
                  '19': 'Conv4-1',
                  '21': 'Conv4-2',
                  '28': 'Conv5-1'}
```

```
features = {}
x = image
```

Contains all the modules of a model

```
for name, layer in model._modules.items():
    x = layer(x)
    if name in layers:
        features[layers[name]] = x
return features
```

```
def gram_matrix(tensor):
    b, d, h, w = tensor.size()
    tensor = tensor.view(b*d, h*w)
    gram = torch.mm(tensor, tensor.t())
    return gram
```

```
# finally!
content_features = get_features(content, vgg)
style_features = get_features(style, vgg)

style_grams = {layer: gram_matrix(style_features[layer])
for layer in style_features}
```

better initialize with content img

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

```
style_weights = {'Conv1_1' : 1.0,
                 'Conv2_1' : 0.75,
                 'Conv3_1' : 0.2,
                 'Conv4_1' : 0.2,
                 'Conv5_1' : 0.2 }
```

Content_weight = 1 ⟶ alpha

style_weight = 1e6 ⟶ beta

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

for _ in range(steps):
    target_features = get_features(target, vgg)
    content_loss = torch.mean(
                    (target_features['conv4_2']
                    - content_features['conv4_2'])
                    **2)
```

```
style_loss = 0
for layer in style_weights:
    target_features = target_features[layer]
    target_gram = gram_matrix(target_feature)
    _, d, h, w = target_feature.shape
    style_gram = style_grams[layer]
    layer_style_loss =
        style_weights[layer] *
        torch.mean((target_gram -
                    style_gram)**2)
    style_loss += layer_style_loss/(d*h*w)

total_loss = content_weight *
             content_loss +
             style_weight *
             style_loss
```

update the target image:

```
optimizer.zero_grad()
total_loss.backward()
optimizer.step()
```

* target is our final output.