AutoArt

I - Abstract

Through this project, we aim to generate more appealing images through neural style transfer. Since the original paper was published [1], there have been multiple extensions that improve the accuracy and flexibility of style transfer. We proposed and implemented two novel methods to improve our outputs for neural style transfer: (i) fine-tuning the model as a classification for a particular style, and (ii) flattening the layers to allow us the convenience of adding style and content loss inside the blocks. Through network dissection, we compare the best positions for style and content loss. We compare various methods of doing style transfer by judging the output stylized images on various quantitative (style loss and content loss) metrics. Among all the traditional NST models, we find mobilenetv2 outperforms all others, hence we perform all experiments with it. In comparison among different mobilenetv2 models, the one that has flattened layers with the fine-tuned model, where the style losses are in the initial layers and content losses are throughout, works the best visually and also has the lowest style and content loss. This can be corroborated by network dissection outputs for activations in initial layers vs final layers.

II - Introduction

In this project, we applied neural style transfer to a variety of input images with different impressionist style images, to generate impressionist versions of the input images. Backbones, such as VGG-19, MobileNetV2, ResNext, and Alexnet, are trained on image classification datasets but are repurposed to do neural style transfer. These out-of-the-box solutions do a decent job of generally transferring any style to any input image, but there are many cases where they fail to apply a style in an aesthetically pleasing way. AutoArt aimed to fine-tune MobileNetV2 specifically for impressionist style transfer to more consistently generate aesthetically pleasing output images.

In addition to fine-tuning, we ran a quantitative analysis on selected MobileNetV2 variants we generated. This included SSIM to score similarities between generated output images and the style and content images that generated them. We used network dissection to visualize how our changes to MobileNetV2 affected each layer's activation. Using this analysis we could better place our content and style loss and understand which of our changes were having the most effect.

III - Related Work

As described in the neural style transfer track, we based most of our development on [1]. [1] explains how style transfer between images is possible because of a new image representation derived from convolutional neural networks (CNN), that allows style and content to be separated. We compared different backbones and selected MobileNetV2 [2] to do a

deeper analysis on. We focused on network dissection [3] for MobileNetv2, and moved content and style loss function placement based on network dissection results.

[2] describes the architecture of the MobileNetv2 backbone we plan to use. It goes into detail on some of the more specific/advanced components of the cutting edge CNN backbones (things like residual blocks and linear bottlenecks). [9] explains why residual blocks are important, to avoid vanishing gradients in deep architectures and to improve a more complicated architecture's ability to model a simple (i.e. identity) function.

[4] and [7] similarly explain the architecture and use cases for ResNext. Unlike VGG-19 or AlexNet [6], these newer architectures use blocks to encapsulate more complicated groups of operations. This can either be to improve efficiency or accuracy (or both).

There are a few other backbones we've been comparing the outputs of, VGG-19 [1] [5], AlexNet [6] and ResNext [4], that we won't be doing a larger analysis of, these were just to give ourselves more output comparisons and a better baseline understanding of common backbone architectures.

Network dissection [3] [10] aims to quantify "the interpretability of latent representations of CNNs but evaluating the alignment between individual hidden units and a set of semantic concepts" [3]. The authors of [3] created a dataset (Broden) and metrics specifically for object classification scoring. Their Intersection over Union (IoU) score is less applicable to our use case, but because MobileNetV2 is trained on ImageNet for classification, we can still apply their visualizations and semantic classification to try to understand what the layers of our models are activating on, and how we should tune our model to potentially improve results. The process for their data segmentation and semantic classification relies on ResNet50 [10] and UPerNet [11] in a Feature Pyramid Network [12] configuration. This allows the model to generate feature maps with ResNet on the bottom-up side, and their UPerNet model to restore resolution while including semantic information. This classification was less important to our project analysis since they did not have texture implemented specifically, it was difficult to use the results from this part to analyze our architecture.

IV - Implementation

Summary

We have repurposed a PyTorch tutorial [8] that originally uses VGG-19 to perform Neural Style Transfer. In this approach, two images are considered - style image and content image. Our goal is to take an input image (which may be initialized randomly), merge these two images with the input image in a way that the input image is transformed into a stylized version of the content image. This is done by minimizing two loss functions: style loss and content loss. Note that this is very different from superimposing the two images - neural style transfer aims to create a new image from the two input images rather than a basic superimposition.

Loading the images

As we are using PyTorch tensors to store the image values, we had to perform some reshaping and value manipulation of the pixel values of the images.

- Reshaping the images to tensors of a particular size ([1, 3, 512, 512]) so that both style and content tensors have the same shape.
- Adding an extra 'fake' dimension to the tensors used as a batch dimension. This was required to fit the network's input dimensions.
- Scaling down the pixel values from a range of [0, 255] to [0,1]. This was done because the pytorch models (torchvision.models) expect a value in this range.
- Finally, the images were loaded into PyTorch tensors.
- On reconverting the tensors into PIL images and printing them out, we observed that the output images were a bit blurry than the original images due to downsampling.
- The images get downsampled further when running on CPU vs when running on GPU. This has been documented in our findings.

Loss Functions

We are trying to minimize two loss functions:

1. Style loss
2. Content loss

Content Loss

We define D_c, which measures how different the content is between the content image and input image. For defining content loss for a particular layer, we define the following terms:

F_{XL} = feature vector of features in layer L of input image X
F_{CL} = feature vector of features in layer L of content image C
w_{CL} = weights
Content Loss = $w_{CL} * || F_{XL} - F_{CL} ||^2$

Content loss is the weighted mean squared difference between the features of the input image X and the content image C.

Style Loss

We define D_s, which measures how different the style is between the style image and input image. For defining style loss for a particular layer, we define the following terms:

F_{XL} = feature vector of features in layer L of input image X
G_{XL} = normalized Gram Matrix of a reshaped version of F_{XL}:
F_{SL} = feature vector of features in layer L of style image S
G_{SL} = normalized Gram Matrix of a reshaped version of F_{SL}
w_{SL} = weights
Gram matrix is used instead of directly using F_{XL} matrix
Style Loss = $w_{SL} * || G_{XL} - G_{SL} ||^2$

Why Gram Matrix?

A Gram matrix is the multiplication of a matrix by its transpose. The Gram matrix is used to find the correlation between features, since we have feature vectors we can take their dot product to get their correlation. Features that are closer together result in closer vectors, which produce a larger dot product. Style can be thought of as features with high co-occurrence between channels. This high co-occurrence is exactly what the Gram matrix captures, this degree of correlation between the channels is used to approximate the style itself, thus allowing us to “separate” style from content.

Lower levels of the CNNs capture information about individual pixel values, whereas higher levels of the CNN capture content. This has informed our placement of the content loss function, as we see many of these implementations place content loss in higher levels. Style loss is typically placed throughout the CNN, since it is measured by the correlation of feature maps in a layer. The style loss is essentially matching the distribution of features between the style image and the output image (in a Gram matrix), so similar features are accounted for and won’t affect the style loss.

Importing the Model

For the initial comparison of the style transfer output, we are importing the following pre-trained neural network backbones, as defined in the PyTorch torchvision module:

1. VGG-19
2. MobileNetV2
3. ResNext
4. AlexNet

While importing the models, we obtain the layer information of the model, which will be stored so that we are able to add the style and content loss functions after certain layers and compare the outputs.

Gradient Descent

We are using the L-BFGS algorithm for gradient descent. The benefits of using this algorithm are:

- Memory efficient
- Fast convergence
- No requirement for tuning the learning rate
- Effective handling of non-convex objectives

Fine-Tuning

We created our dataset where impressionist style images (link) are positive images and non-impressionist style images (link) are negative images. We used a binary classification task to train mobilenetv2. For the mode, we changed the last classification layer of mobilenetv2 to give 2 outputs only. Then we use this fine tuned model for neural style transfer.

Network Dissection

Network dissection has been done on a variety of backbones to quantify how each layer contributes to object classification. Our goal was not object classification, so many of the metrics

used could not be directly applied to our MobileNetV2 variants. The focus of our network dissection was to visualize what each layer was generally activating on (since it varies from unit to unit).

Depending on dataset size, this part was very computationally intense, so it was set up to run on one of our personal GPUs. Moving datasets around on Google Drive/Colab is cumbersome and can lead to quicker compute resource timeout, which also meant this was much easier to run on a personal computer. The set up time for this was more than expected, especially since the network dissection work was completed in 2020 and has not been updated. After getting a PC setup with Ubuntu and getting CUDA installed, many updates needed to be made to outdated code in addition to creating new datasets for network dissection.

We created our own experiment Python script that can load our 4 variants (MobileNetV2 pre-trained, MobileNetV2 fine-tuned, flattened MobileNetV2 pre-trained, and flattened MobileNetV2 fine-tuned) with our custom datasets. The code was stored in a Github repository.

We did a lot of experimenting with what set of images we should visualize activations on, to get an idea of how texture (style) might activate, since both the backbone we're using and the analysis we're doing are based on object classification. We tried images with the same content but different styles (i.e. Mona Lisa original, cartoon, abstract, etc). We compared activations on input, content and output triplets as well as our training dataset and new impressionist and not impressionist style images. The input, content and output triplets gave us some context for what style vs content is activating, so we focused on our analysis on these.

We also ran the object classification statistics, just to get an idea of how our changes were affecting the model. We could see when object classification was getting worse, but couldn't directly tie that to style transfer being worse. It would make sense that because we're changing the goal of this model but still using object classification metrics, that it would score worse. If a model does much worse on object classification, we could interpret that as a potential decrease in neural style transfer performance, since we do need our model to be able to do some amount of object detection in order to separate style and content.

SSIM

SSIM is a structural similarity index used for measuring image quality. The original paper which introduced SSIM [13] states that when comparing images, the mean squared error (MSE) doesn't explain perceived similarity as observed by a human. Structural similarity aims to address this shortcoming by taking luminance, contrast and structure of the image into account. SSIM(image1, image2) outputs a value between 0 and 1. A value closer to 1 indicates high perceived similarity between the two images.

V - Experiments and Results

Hyperparameter Tuning

We finetune the hyperparameters after fixing the positions for style loss and content loss. We increased the style weight to have more style impact on the output image. When starting with a randomly generated image, having a small content loss leads to moving the entire image towards the style, however when we increase the content loss from 10 to 100, the model generates an output which has a huge impact from the content image, helping in recreating the content image. When we increase the number of epochs for alexnet, we get better results with a smaller loss for both style and content, however with a much smaller epochs for other architectures the model converges.

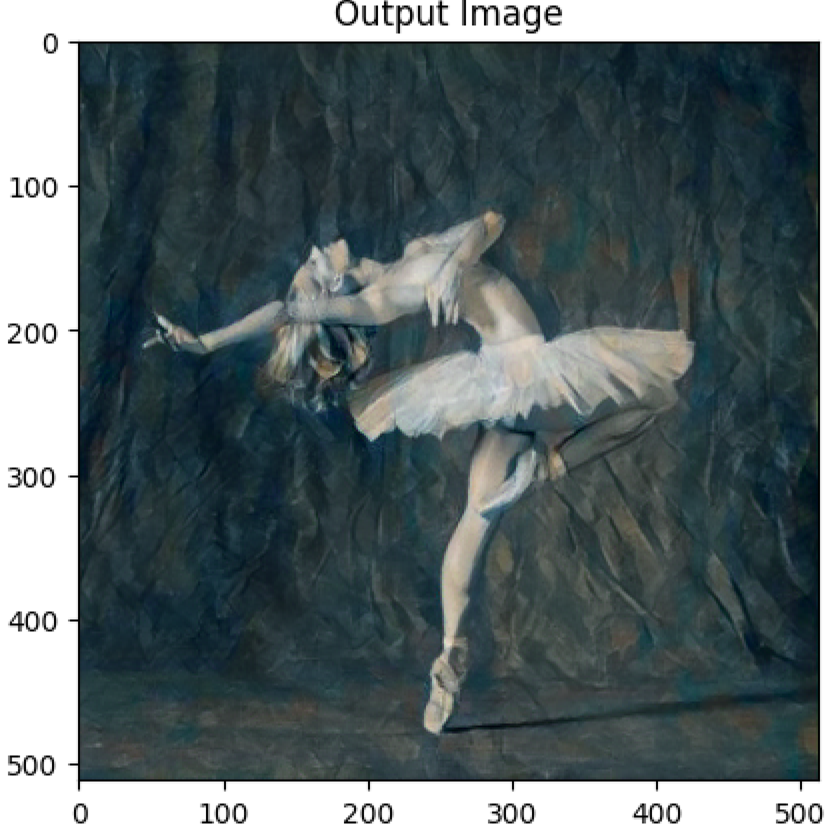
We also vary the placement of the style and content loss and observe visually interesting results for the various placements. We corroborate our findings with network dissection.

VGG-19

As mentioned in [5], there is a tradeoff between convolutional neural networks depth and accuracy. Having 16-19 weight layers gives the best output.
On printing the structure of the VGG-19 model, we can see that it is one Sequential block with 2D convolutions, max pooling and ReLU layers. In our implementation, we have currently added style loss and content loss after the following layers:

```
content_layers_default = ['conv_4']
style_layers_default = ['conv_1', 'conv_2', 'conv_3', 'conv_4',
'conv_5']
```

Content weight: 1
Style weight: 1000000
Results:
After 300 runs, we get the following loss values:
Style Loss : 0.266553 Content Loss: 2.349637



MobileNetV2

Currently, while implementing MobileNetV2, we have flattened each of the sequential blocks. That is, originally the structure was:

```
Sequential(
  (0): Conv2dNormActivation(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU6(inplace=True)
  )
  (1): InvertedResidual(
    (conv): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU6(inplace=True)
      )
      (1): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (2): InvertedResidual(
    (conv): Sequential(
```

We converted ('flattened') this structure to the following:

```
Sequential(
  (conv_0_0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
  (bn_0_1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu_0_2): ReLU6(inplace=True)
  (conv_1_0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False)
  (bn_1_0): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu_1_0): ReLU6(inplace=True)
  (conv_1_1): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn_1_2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv_2_0): Conv2d(16, 96, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn_2_0): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu_2_0): ReLU6(inplace=True)
  (conv_2_1): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), groups=96, bias=False)
  (bn_2_1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu_2_1): ReLU6(inplace=True)
  (conv_2_2): Conv2d(96, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn_2_3): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv_3_0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn_3_0): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

In the original structure, we would only have been able to introduce style and content losses after each of the blocks. However upon flattening the structure, we now have the flexibility to introduce style and content losses after each individual convolution layer. However, it is no longer MobileNetV2 but rather a modified structure.

Upon further reading, we have found that flattening this structure may not be beneficial and could have adverse effects. MobileNetv2 introduced inverted residual with linear bottleneck. Bottleneck layers are used because it was assumed that manifolds of interest would be embedded in low-dimensional spaces. So reducing the dimensionality allows the manifold of interest to occupy the entire space, and gives good accuracy while reducing computations. However, nonlinear transforms (ReLU) break this. When ReLU collapses the channel it loses information in that channel. The linear bottleneck is needed to prevent nonlinearities from destroying information. (ReLU is capable of preserving complete information, but only if the input lies in a low-dimensional subspace).

One of our future goals is to compare the output of our current implementation with the output of the original structure of MobileNetV2.

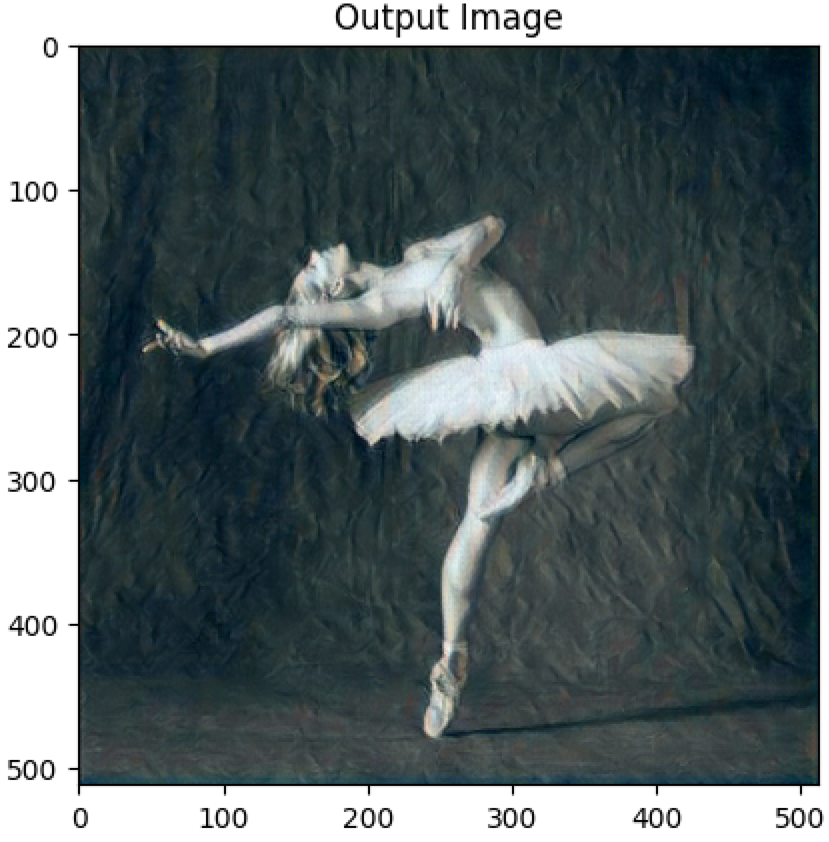
Content weight: 1

Style weight: 10000000

Results:

After 300 runs, we get the following loss values:

Style Loss : 0.003722 Content Loss: 0.109142



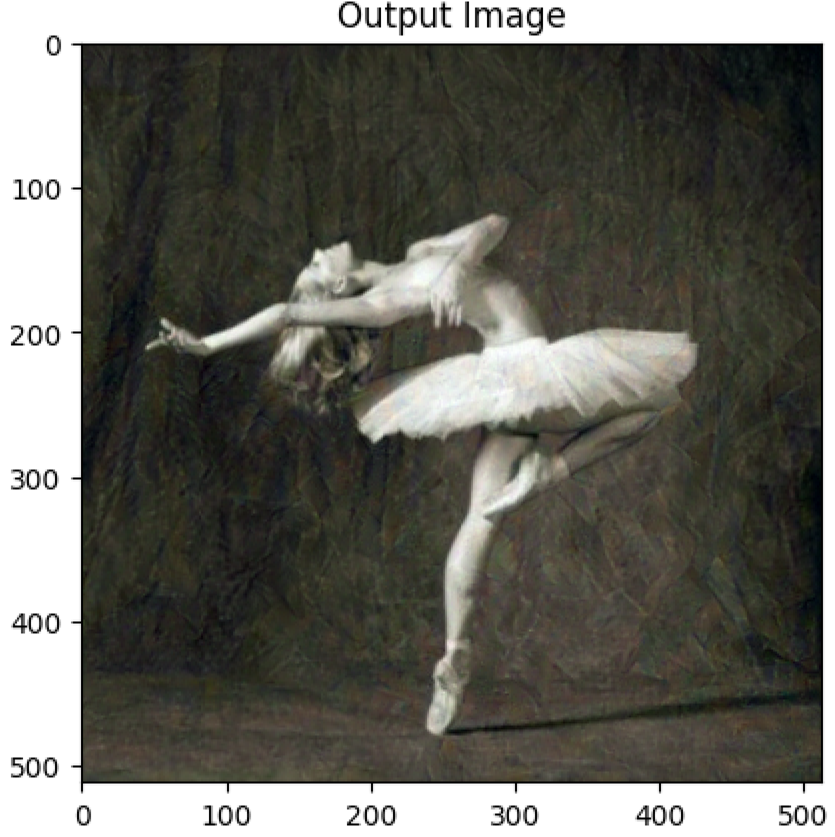
ResNext

Similar to MobileNet, we have tried to flatten the blocks in the network to add flexibility to the placement of style and content loss functions. However, due to having residual connections in the model, we cannot flatten the structure. This is the same reason as why we cannot flatten MobileNetV2.

Currently we have placed the loss functions after the following layers:

```
content_layers_default = ['layer2']
style_layers_default = ['layer1', 'layer2', 'layer3', 'layer4']
```

Content weight: 1
Style weight: 1000000
Results:
After 300 runs, we get the following loss values:
Style Loss : 0.059025 Content Loss: 0.011729

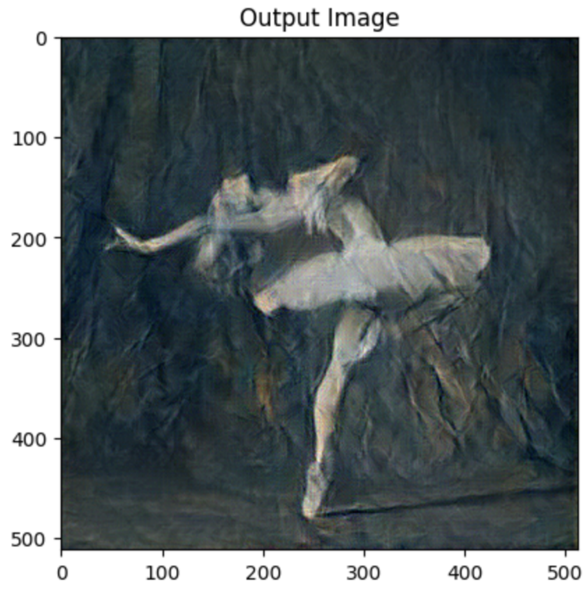


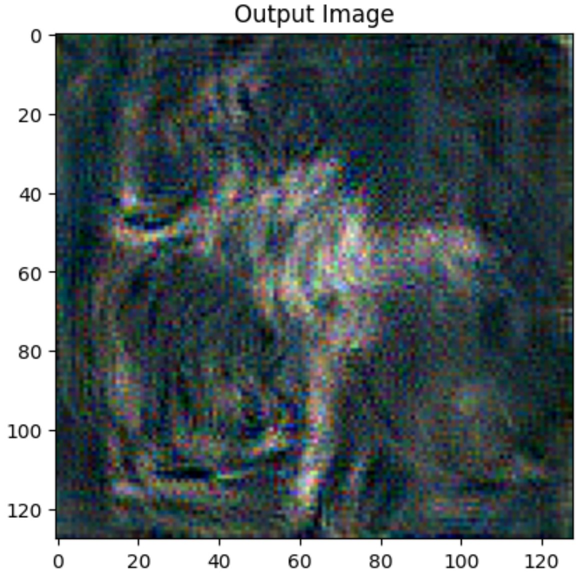
AlexNet

Content weight: 10000000
Style weight: 100
Results:
After 1000 runs, we get the following loss values:
Style Loss : 84.393143 Content Loss: 317.107697

Running on CPU vs GPU:
The images get downsampled further when running on CPU vs when running on GPU. This has been documented in our findings for AlexNet: when we run the network on CPU, we get a much higher loss value and a poorly stylized content image as the output.
When running the network on CPU, after 1000 runs, we get the following loss values:
Style Loss : 709.549011 Content Loss: 2085.458740

This is because downsampling the image reduces the features in the image, making it further difficult to minimize the loss to the same extent as the higher resolution image.





Left: Running AlexNet on GPU

Right: Running AlexNet on CPU

VGG-19, MobileNetv2 Comparison

In the VGG-19 paper [5], the authors investigated the effect of network depth on accuracy, for an image recognition problem. They use 3x3 convolution filters with 16-19 layers to produce a model that achieves high accuracy and generalizes to other datasets. The stack of smaller receptive fields allows for also stacking more nonlinear rectification layers, which improves feature detection. VGG-19 is not compute or memory efficient though (as mentioned in class).

MobileNetV2 does improve on efficiency and accuracy, when compared to VGG-19. MobileNetv2's architecture is similar to ResNet's, as it uses residual units and bottlenecks. However instead of 3x3 convolution it uses depthwise convolution.It uses a different bottleneck architecture to increase the channel dimensions for this depthwise convolution.

Throughout the subsequent findings, we have used MobileNetV2 as it consistently achieved the lowest loss values.

Comparing various content images

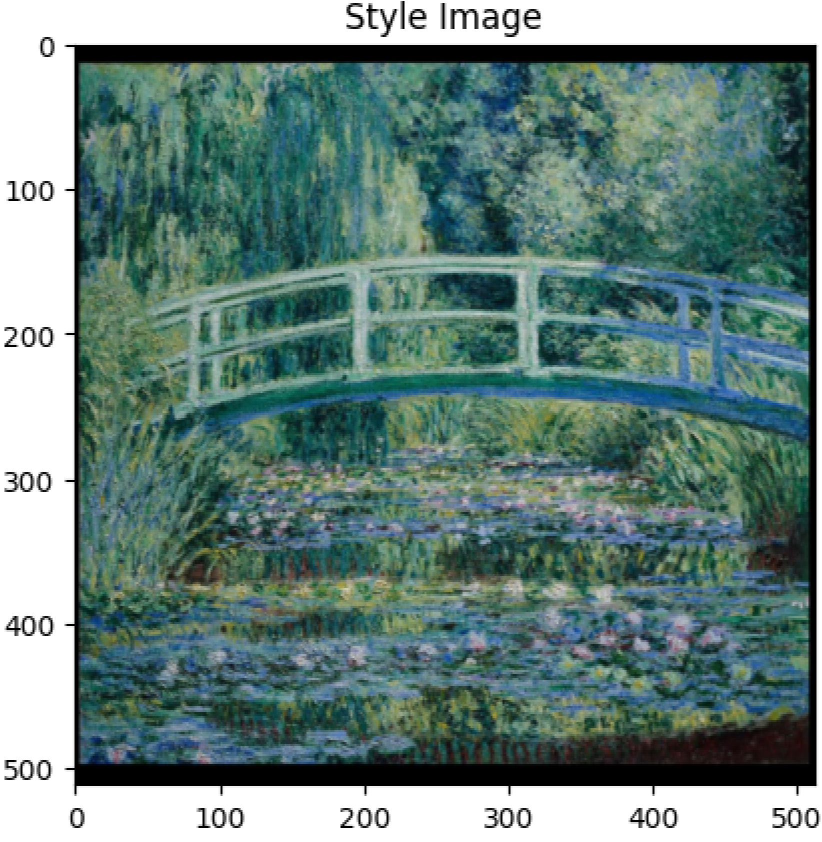
Here, we used flattened, fine-tuned Mobilenetv2 to perform neural style transfer on various type of content images. We ran the model with the following hyperparameters:

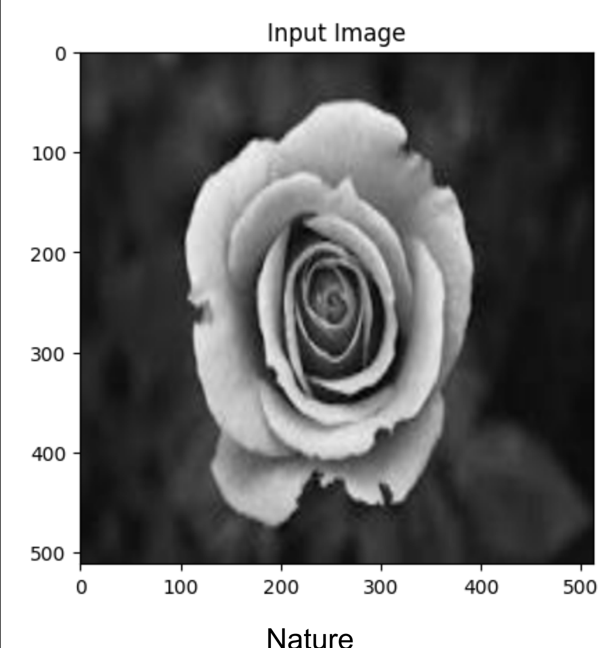
```
style_weight=100000, content_weight=1, num_steps=500
```

These are the loss placement layers for style and content loss:

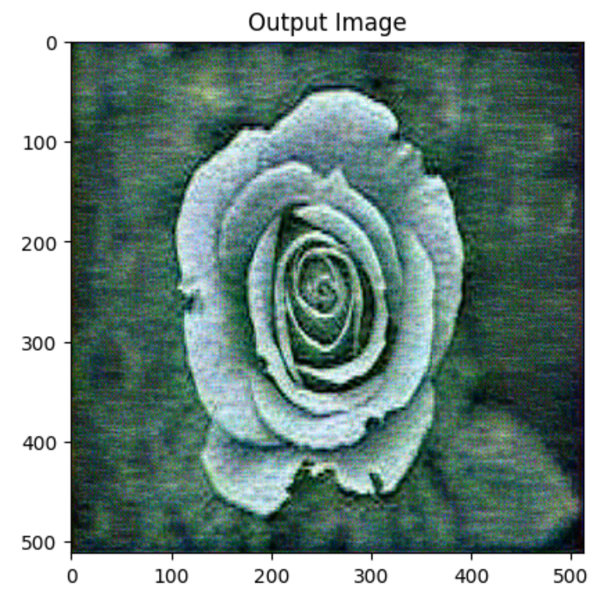
```
content_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2',
'conv_8_2', 'conv_14_2']
style_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2',
'conv_8_2', 'conv_14_2']
```

This is the style image as used throughout the analysis:

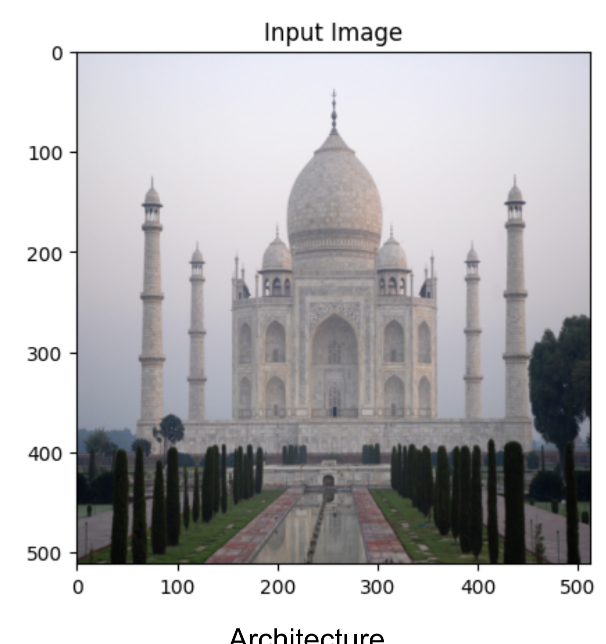




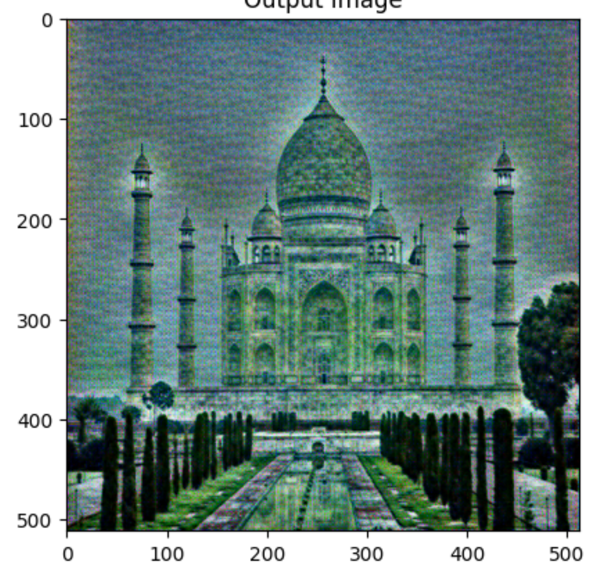
Nature



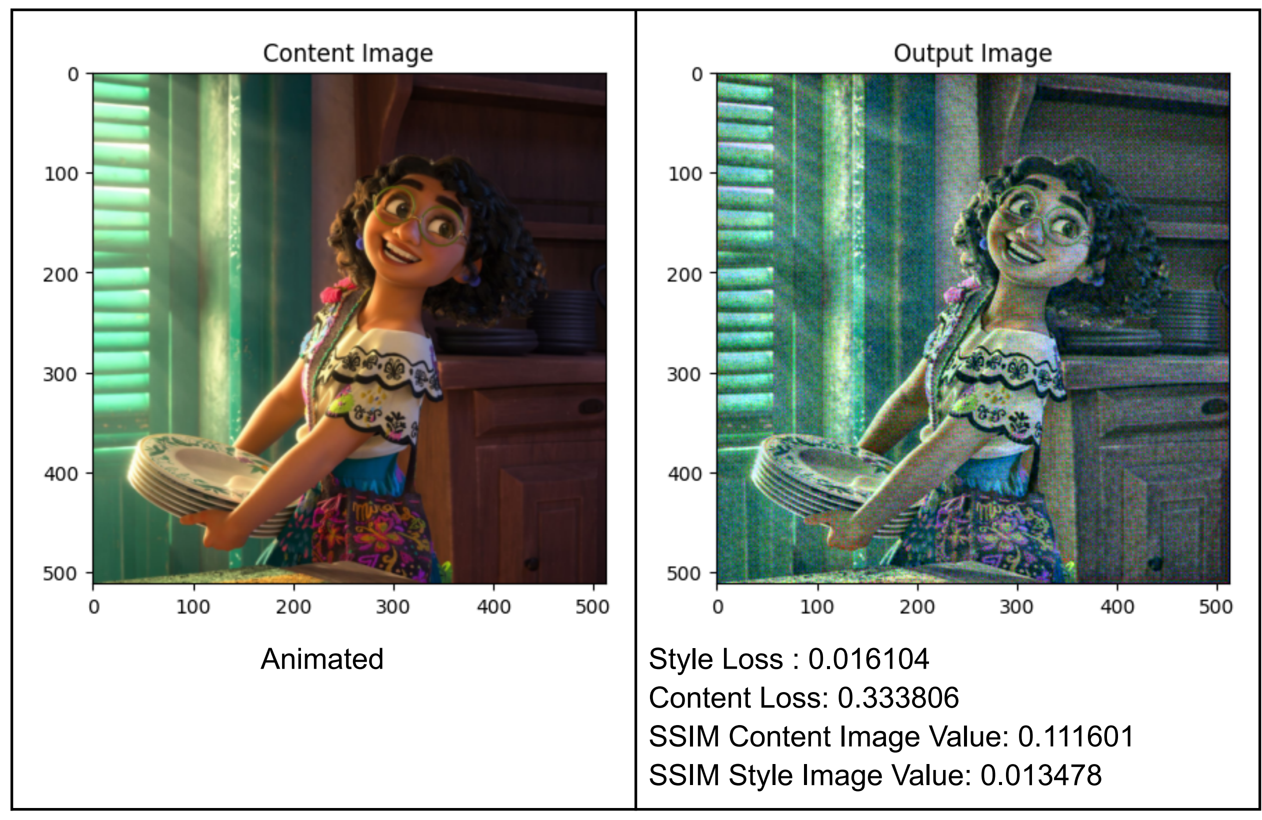
Style Loss : 0.068051 Content Loss: 0.560982
SSIM Content Image Value: 0.063249
SSIM Style Image Value: 0.011822



Architecture



Style Loss : 0.019836
Content Loss: 0.455306
SSIM Content Image Value: 0.095920
SSIM Style Image Value: 0.011825

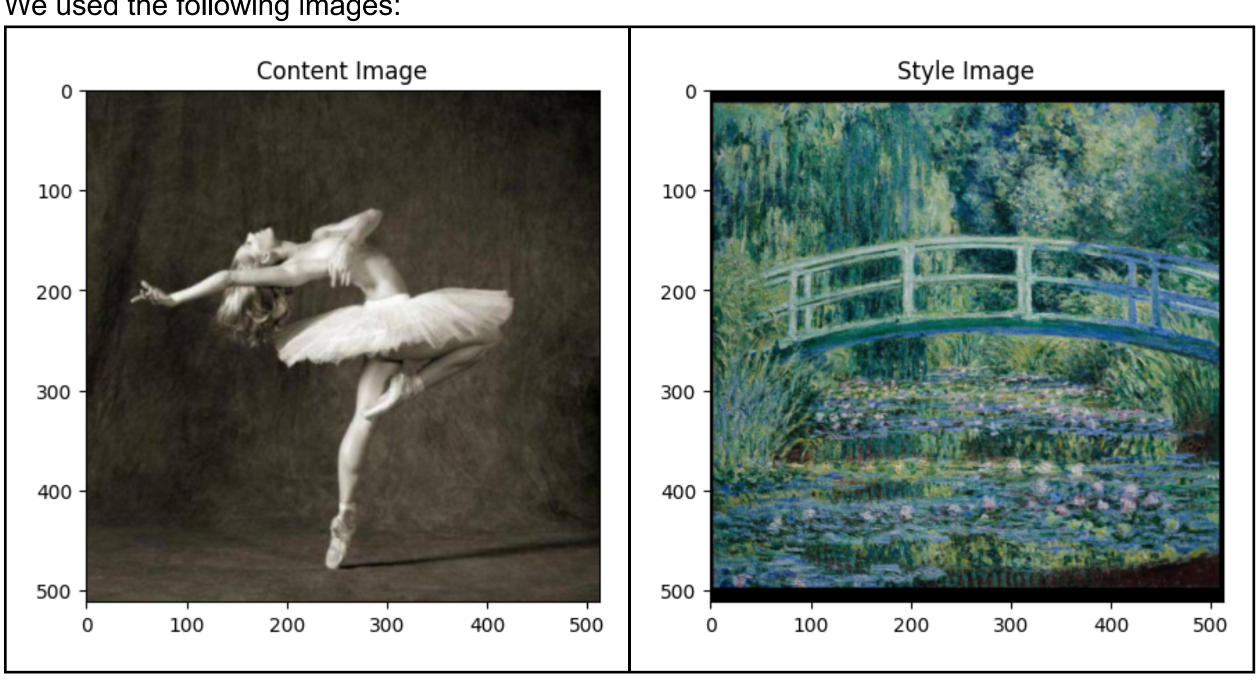


Animated

Style Loss : 0.016104
Content Loss: 0.333806
SSIM Content Image Value: 0.111601
SSIM Style Image Value: 0.013478

Hyperparameter Tuning

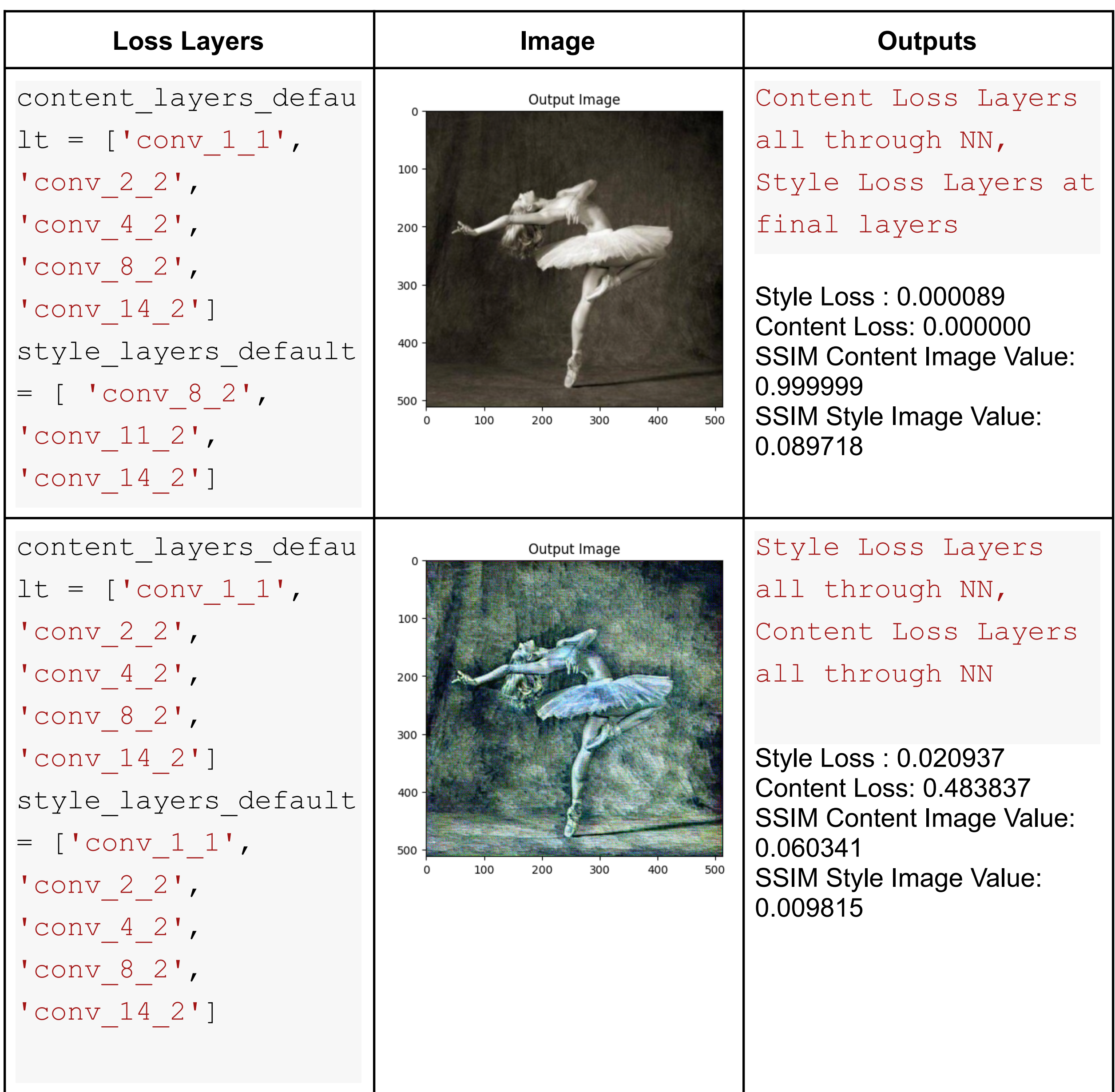
We used the following images:



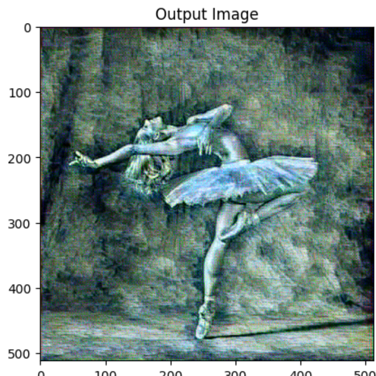
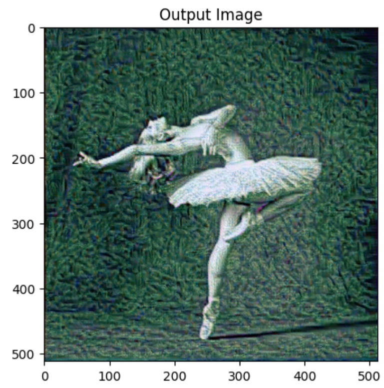
Tuning the Layers for Content and Style Loss

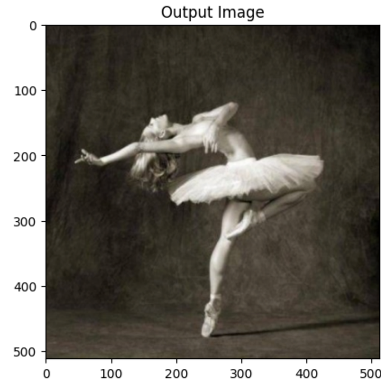
We ran the neural style transfer on the flattened, fine-tuned network and got the following results by modifying the layers on which style and content loss is measured. We used the following hyperparameter values:

```
style_weight=100000, content_weight=1, num_steps=500
```



Loss Layers	Image	Outputs
`content_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2', 'conv_8_2', 'conv_14_2']` `style_layers_default = [ 'conv_8_2', 'conv_11_2', 'conv_14_2']`	Output Image	`Content Loss Layers all through NN, Style Loss Layers at final layers` Style Loss : 0.000089 Content Loss: 0.000000 SSIM Content Image Value: 0.999999 SSIM Style Image Value: 0.089718
`content_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2', 'conv_8_2', 'conv_14_2']` `style_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2', 'conv_8_2', 'conv_14_2']`	Output Image	`Style Loss Layers all through NN, Content Loss Layers all through NN` Style Loss : 0.020937 Content Loss: 0.483837 SSIM Content Image Value: 0.060341 SSIM Style Image Value: 0.009815

`content_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2', 'conv_8_2', 'conv_14_2']` `style_layers_default = ['conv_1_1', 'conv_2_2']`	Output Image	`Content Loss Layers all through NN, Style Loss Layers at initial layers` Style Loss : 0.036559 Content Loss: 0.518320 SSIM Content Image Value: 0.065759 SSIM Style Image Value: 0.013625
`content_layers_default = ['conv_1_1', 'conv_2_2']` `style_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2', 'conv_8_2', 'conv_14_2']`		`Content Loss Layers at initial layers, Style Loss Layers all through NN` Style Loss : 0.015872 Content Loss: 0.389904 SSIM Content Image Value: 0.064501 SSIM Style Image Value: 0.010989
`content_layers_default = ['conv_8_2', 'conv_14_2']` `style_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2', 'conv_8_2', 'conv_14_2']`		`Content Loss Layers at final layers, Style Loss Layers all through NN` Style Loss : 0.000015 Content Loss: 0.000044 SSIM Content Image Value: 0.293853 SSIM Style Image Value: 0.046641

`content_layers_default = ['conv_8_2', 'conv_11_2', 'conv_14_2']` `style_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2']`	Output Image	`Content Loss Layers at final layers, Style Loss Layers at initial layers` Style Loss : 0.000001 Content Loss: 0.000039 SSIM Content Image Value: 0.293300 SSIM Style Image Value: 0.046930
`content_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2']` `style_layers_default = ['conv_8_2', 'conv_11_2', 'conv_14_2']`		`Content Loss Layers at initial layers, Style Loss Layers at final layers` Style Loss : 0.000089 Content Loss: 0.000000 SSIM Content Image Value: 0.999999 SSIM Style Image Value: 0.089718

Fine Tuned Flattened vs Fine Tuned Unflattened (Original) MobileNetV2

We used the following hyperparameter values:

```
style_weight=10000, content_weight=1, num_steps=100
```

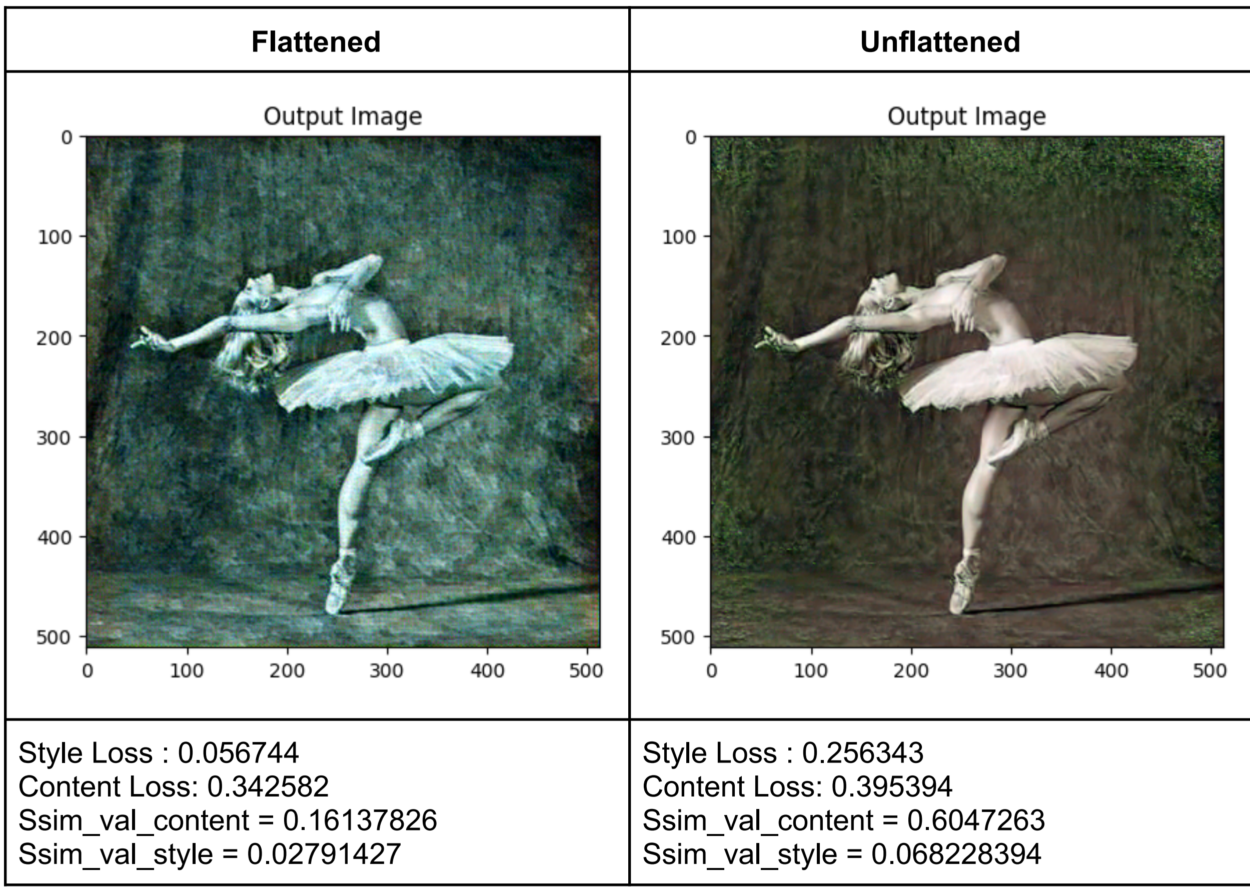
```
Unflattened:
```

```
content_layers_default = ['layer_1', 'layer_2', 'layer_4',
'layer_8', 'layer_14']
style_layers_default = ['layer_1', 'layer_2', 'layer_4',
'layer_8', 'layer_14']
```

```
Flattened:
```

```
content_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2',
'conv_8_2', 'conv_14_2']
```

```
style_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2',
'conv_8_2', 'conv_14_2']
```



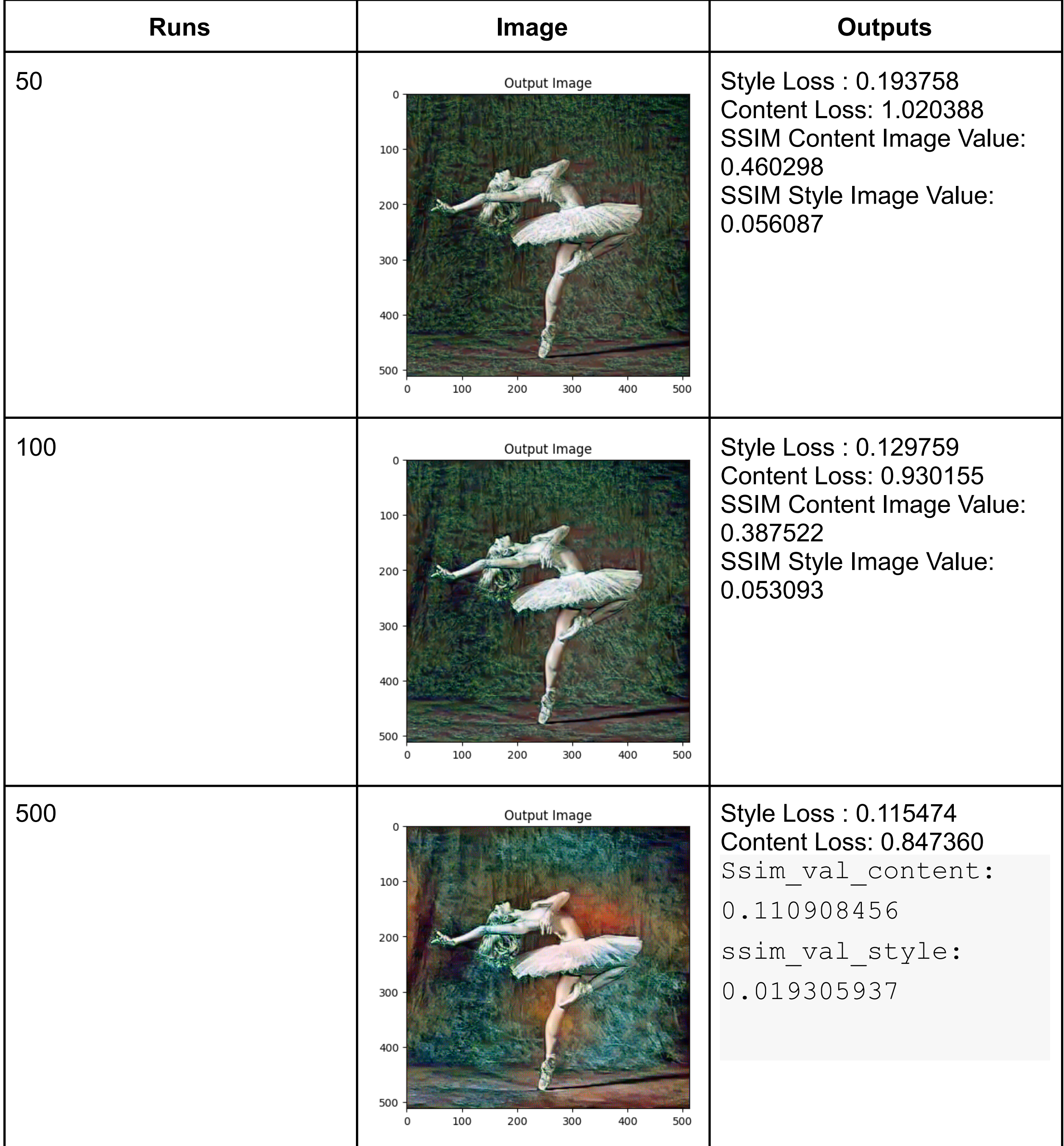
Flattened	Unflattened
Style Loss : 0.056744 Content Loss: 0.342582 Ssim_val_content = 0.16137826 Ssim_val_style = 0.02791427	Style Loss : 0.256343 Content Loss: 0.395394 Ssim_val_content = 0.6047263 Ssim_val_style = 0.068228394

Increasing Number of Runs

Running on Fine-Tuned, Unflattened MobileNet V2:

We used the following hyperparameter values:

```
style_weight=100000, content_weight=1,
content_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2',
'conv_8_2', 'conv_14_2']
style_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2',
'conv_8_2', 'conv_14_2']
```



Runs	Image	Outputs
50		Style Loss : 0.193758 Content Loss: 1.020388 SSIM Content Image Value: 0.460298 SSIM Style Image Value: 0.056087
100		Style Loss : 0.129759 Content Loss: 0.930155 SSIM Content Image Value: 0.387522 SSIM Style Image Value: 0.053093

500		Style Loss : 0.115474
		Content Loss: 0.847360
		`Ssim_val_content: 0.110908456`
		`ssim_val_style: 0.019305937`

Running on Fine-Tuned, Flattened MobileNet V2

Here, we are using the same weights as unflattened for a fair comparison. We used the following hyperparameter values:

```
style_weight=100000, content_weight=1
content_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2',
'conv_8_2', 'conv_14_2']
```

```
style_layers_default = ['conv_1_1', 'conv_2_2', 'conv_4_2',
'conv_8_2', 'conv_14_2']
```

Runs	Image	Outputs
50		Style Loss : 0.031723 Content Loss: 0.506261 SSIM Content Image Value: 0.354319 SSIM Style Image Value: 0.051471
100		Style Loss : 0.022443 Content Loss: 0.473784 SSIM Content Image Value: 0.333108 SSIM Style Image Value: 0.048202
500		Style Loss : 0.019978 Content Loss: 0.492322 SSIM Content Image Value: 0.061220 SSIM Style Image Value: 0.010030

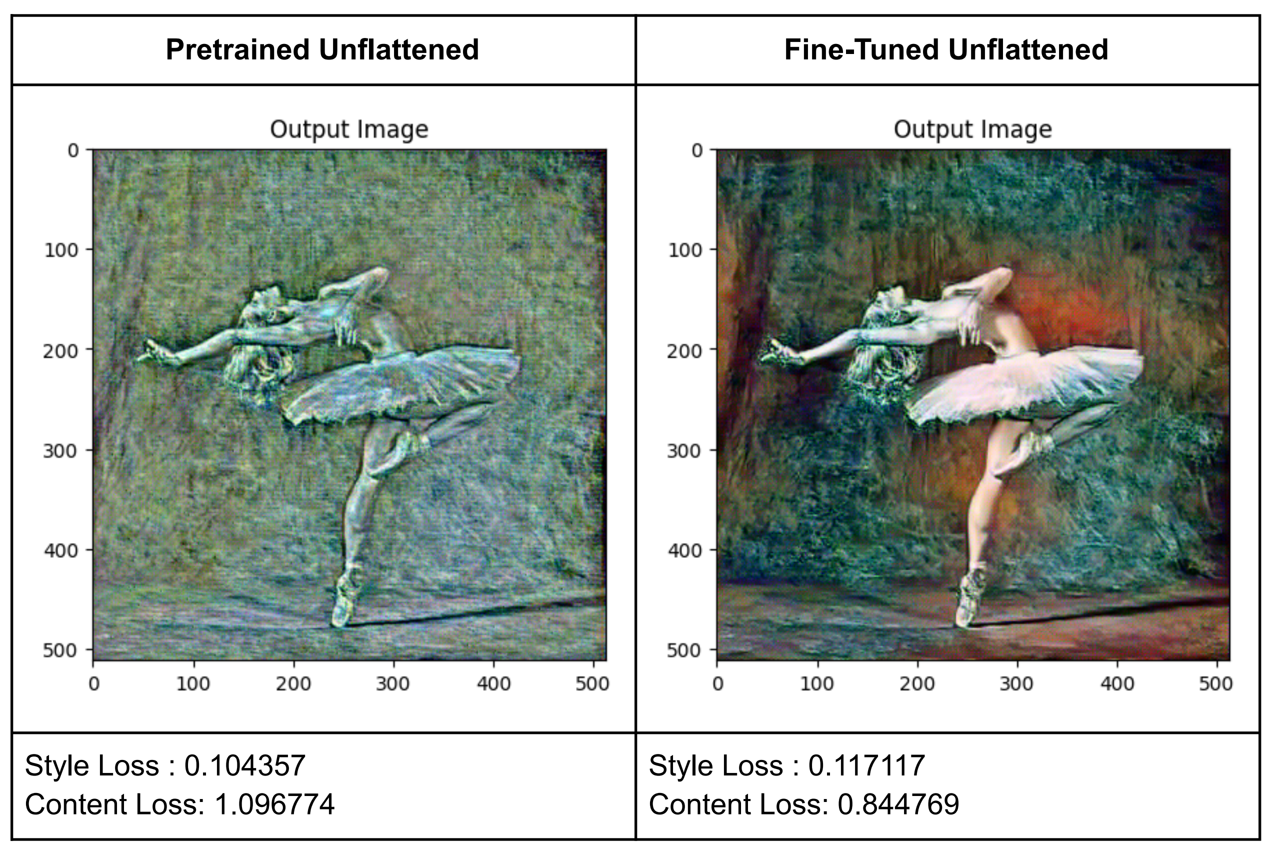
The flattened, fine-tuned model exhibits lower content loss than unflattened fine-tuned model.

Fine-Tuning

Comparison between fine-tuned unflattened and pre-trained unflattened

We used the following hyperparameter values:

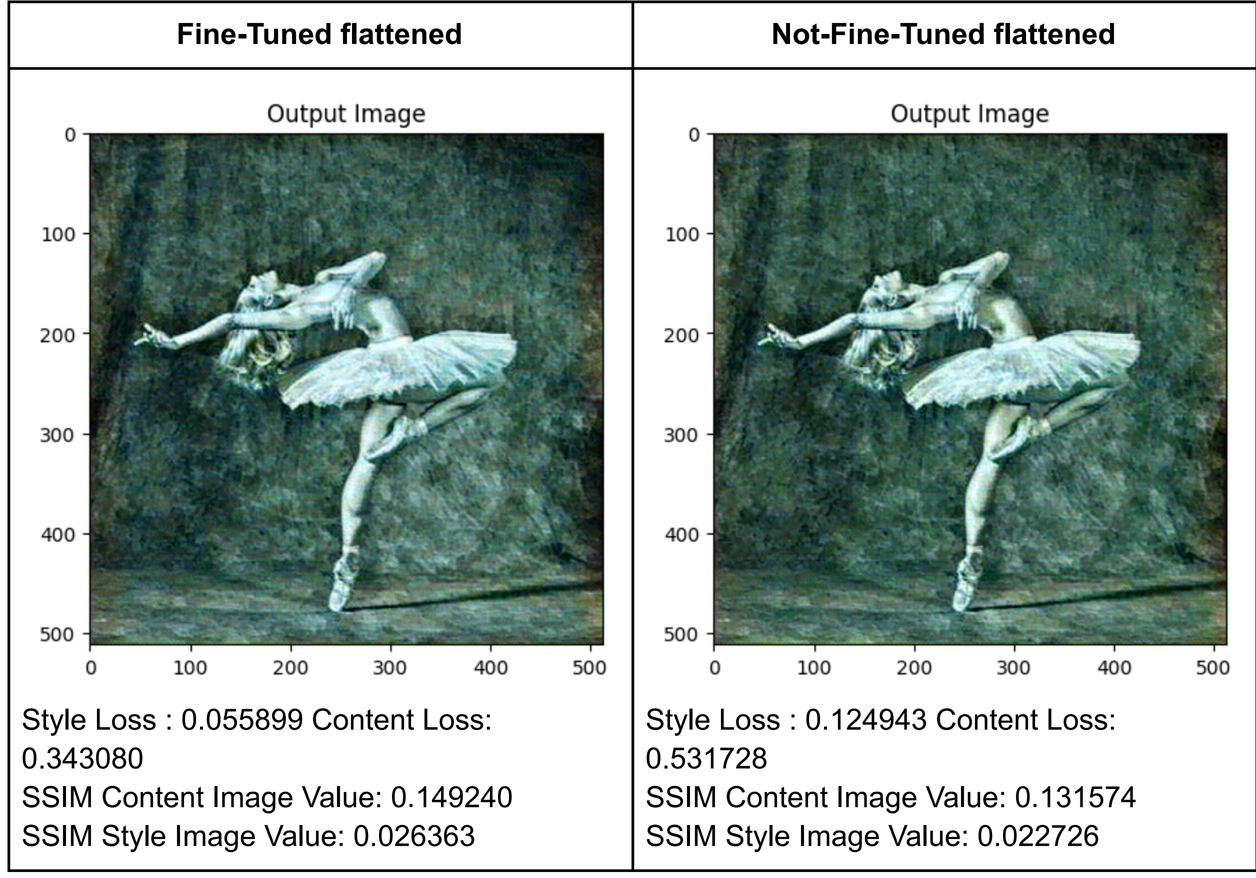
```
style_weight=100000, content_weight=1, num_steps=500
content_layers_default = ['layer_1', 'layer_2', 'layer_4',
'layer_8', 'layer_14']
style_layers_default = ['layer_1', 'layer_2', 'layer_4',
'layer_8', 'layer_14']
```



Pretrained Unflattened	Fine-Tuned Unflattened
Style Loss : 0.104357 Content Loss: 1.096774	Style Loss : 0.117117 Content Loss: 0.844769

Comparison bw fine-tuned flattened and non-fine-tuned flattened

```
style_weight=10000, content_weight=1, num_steps=100
```



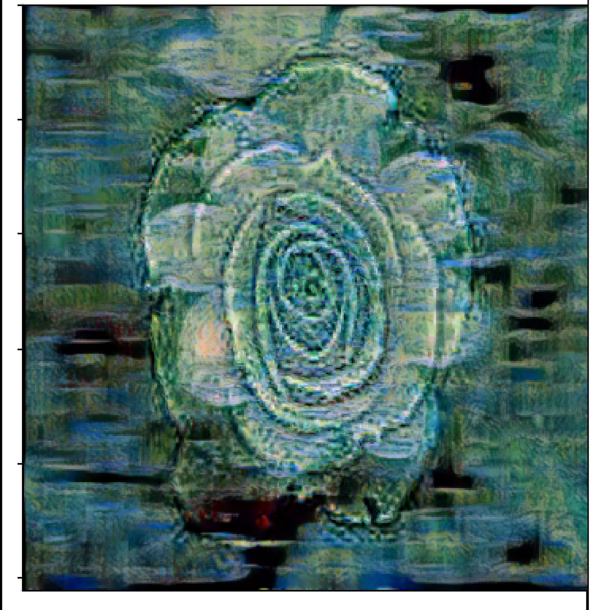
Fine-Tuned flattened	Not-Fine-Tuned flattened
Style Loss : 0.055899 Content Loss: 0.343080 SSIM Content Image Value: 0.149240 SSIM Style Image Value: 0.026363	Style Loss : 0.124943 Content Loss: 0.531728 SSIM Content Image Value: 0.131574 SSIM Style Image Value: 0.022726

VI - Analysis

Flattened fine-tuned MobileNetV2 vs VGG-19

Flattening and fine-tuning our model on impressionist style did show an improvement in our model's ability to apply impressionist style to a content image.

Content image:

VGG-19	MobileNetV2 - flattened and fine-tuned
	

Fine tuned vs not fine tuned

We observed consistent better loss values for the fine-tuned model. This can be attributed to the fact that fine-tuning the model trained the model to work better on doing style-transfer for this particular style (i.e. impressionist art).

Fine tuned flattened vs fine tuned unflattened

We observed generally better (lower) loss values for the flattened MobileNetV2 model. This is because flattening the model gives us more control over the layers we can apply the style and content loss to. Flattening the MobileNetV2 model gets rid of the inverted residual layers and the blocks, which could lead to issues deeper in the architecture with vanishing gradients. This might mean for the flattened variant that we could use a shallower architecture (at least for the small images we're running on) and maintain our lower loss scores. Texture is typically identified at lower layers of the model, which might explain why this model did better at impressionism style transfer.

Tuning the Layers for Content and Style Loss

We observed that the combination of having style loss to initial layers and content loss to final layers gave the lowest loss values, however it did not seem the most visually appealing result. On adding style loss throughout the network and adding content loss to the initial layers, we were able to achieve the subjectively most appealing result. Below are a few observations from the experiments we performed:

1. **Adding Style Loss to the final layers:** The output image had no contribution of the style image - this can be seen as invariant of the fact whether we apply content loss

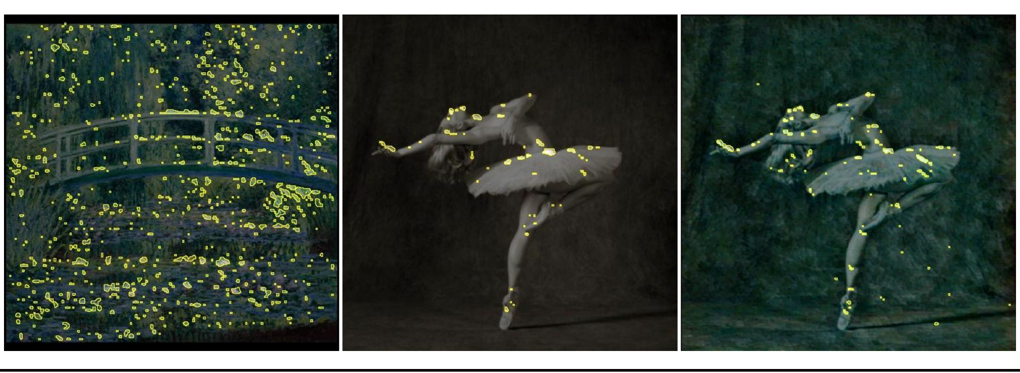
initially or throughout the layers. This can be attributed to the fact that style changes to the image are more on a pixel-by-pixel level, which are the characteristics of the initial layers.

2. **Adding Content Loss Layers at initial layers and Style Loss Layers at final layers:** When there is no overlap between the content and style loss layers - the output image is the same as the content image. The model needs to learn the relationship between content and style and there needs to be a balance between the two, otherwise the model will learn to just output the lowest content loss possible, which is the same image as input.
3. **Adding Content Loss Layers at final layers:** The model performed best and gave the best looking output image when we applied content loss to the initial layers or throughout the network. Based on our network dissection (shown in the tables below), we found that adding content loss to only the final layers in the flattened model meant content loss was not being calculated on activations that were contributing to identifying the content of an image. Because flattening caused more activations earlier and degraded towards the higher layers we saw improvements when we calculated content loss at lower layers.
4. **Adding Content Loss Layers and Style Loss Layers throughout the neural network:** This combination gave a visually appealing output, which leads us to believe that adding both losses throughout the network is an effective way to create visually appealing style transfer images.
5. **Adding style loss throughout the network and adding content loss to the initial layers:** We were able to achieve the subjectively most appealing result. We observed that it is important to add content and style loss especially in the initial layers, as it seems to create an image that is able to better understand the style component. This can be corroborated by the network dissection as shown below.

Network Dissection

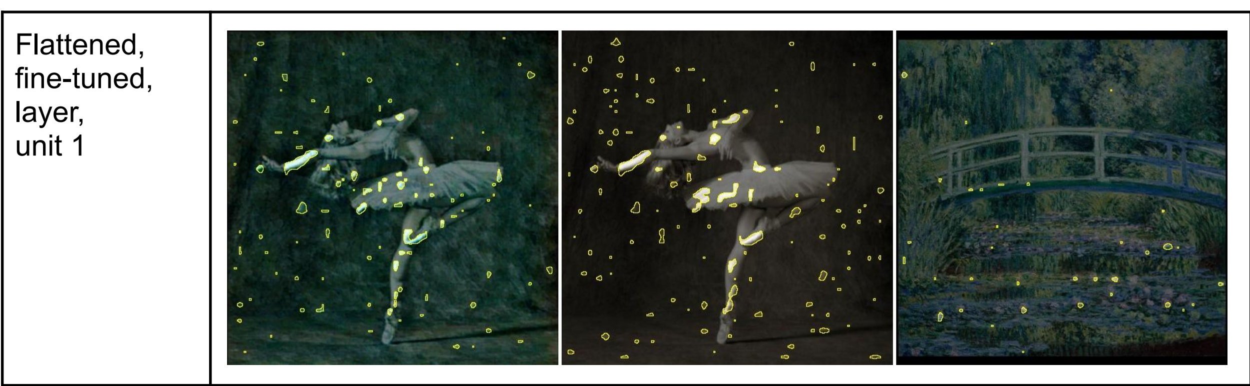
Through network dissection, we can corroborate our observations that adding style loss and adding content loss to the initial layers leads to better visually appealing style transfer images.

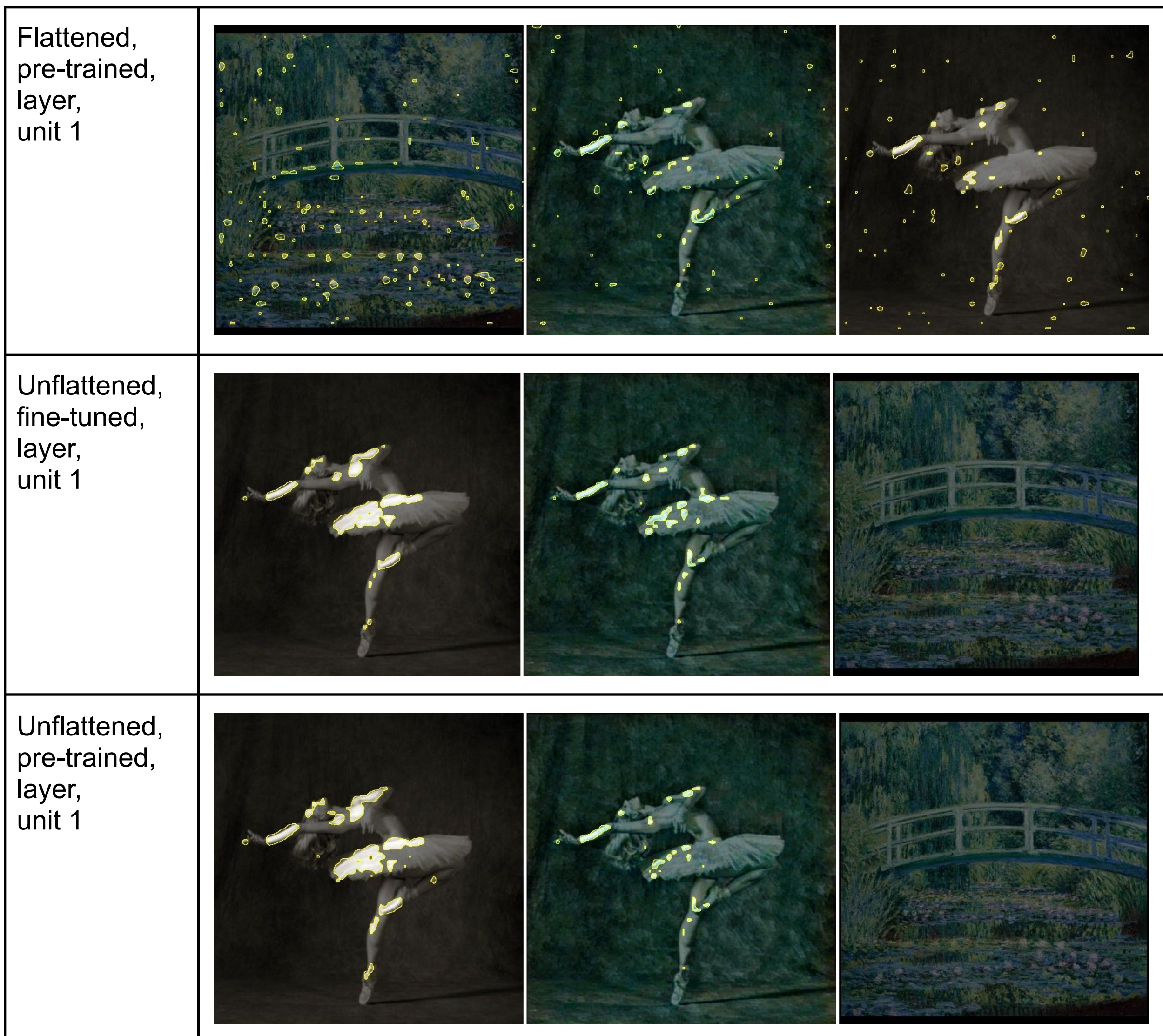
Table showing activations throughout the flattened fine-tuned model

Flattened, fine-tuned, layer 2 unit 2	

Flattened, fine-tuned, layer 4 unit 3	
Flattened, fine-tuned, layer 8 unit 0	
Flattened, fine-tuned, layer 14 unit 14	

Table showing activations of our 4 MobileNetV2 variants





Increasing Number of Runs

Increasing the number of epochs for which the neural network runs consistently improves the performance of the model in terms of style and content loss values.

VII - Conclusion

In this project, we learned how to use convolutional neural networks to do neural style transfer. We learnt about various pre-existing models such as VGG-19, MobileNetV2, ResNext, and Alexnet. We observed that even with low style and content loss values for a particular neural style transfer output, the output image may not appear to be the most visually appealing to a human. This motivated us to fine-tune MobileNetV2 to create a model trained on recognizing a particular art style (impressionism). We also tried flattening the block structure of the model so we could experiment with loss function placements. We observed that we were able to produce more visually appealing outputs with these modifications. We tried various

combinations of hyperparameters and loss placement in the CNN layers, and found visually interesting results as documented. Through network dissection, our findings about placement of loss layers was corroborated. The entire experience of working with each other, TAs and Professor Shi on this engaging project was extremely rewarding and we believe that we have created a novel model with visually appealing results.

References

1. Image Style Transfer Using Convolutional Neural Networks; Leon A. Gatys, Alexander S. Ecker, Matthias Bethge; https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7780634
2. MobileNetV2: Inverted Residuals and Linear Bottlenecks; Mark Sandler Andrew Howard Menglong Zhu Andrey Zhmoginov Liang-Chieh Chen; https://arxiv.org/pdf/1801.04381.pdf
3. Network Dissection: Quantifying Interpretability of Deep Visual Representations; David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, Antonio Torralba; http://netdissect.csail.mit.edu/
4. Aggregated Residual Transformations for Deep Neural Networks; Saining Xie Ross Girshick Piotr Doll´ar Zhuowen Tu Kaiming He; https://arxiv.org/pdf/1611.05431v2.pdf
5. VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION; Karen Simonyan & Andrew Zisserman; https://arxiv.org/pdf/1409.1556.pdf
6. ImageNet Classification with Deep Convolutional Neural Networks; Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton; https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
7. How to Visualize Filters and Feature Maps in Convolutional Neural Networks; Jason Brownlee; https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/
8. PyTorch Tutorial; https://pytorch.org/tutorials/advanced/neural_style_tutorial.html#neural-transfer-using-pytorch
9. Deep Residual Learning for Image Recognition; Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun; https://arxiv.org/pdf/1512.03385.pdf
10. Understanding the Role of Individual Units in a Deep Neural Network; David Bau, Jun-Yan Zhu, Hendrik Strobelt, Agata Lapedriza, Bolei Zhou, and Antonio Torralba; https://arxiv.org/pdf/2009.05041.pdf
11. Unified Perceptual Parsing for Scene Understanding; Tete Xiao, Yingcheng Liu, Bolei Zhou, Yuning Jiang, Jian Sun; https://arxiv.org/pdf/1807.10221.pdf
12. Feature Pyramid Networks for Object Detection; Tsung-Yi Lin, Piotr Doll´ar, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie; https://arxiv.org/pdf/1612.03144.pdf

13. Image Quality Assessment: From Error Visibility to Structural Similarity; Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, Eero P. Simoncelli; https://www.cns.nyu.edu/pub/eero/wang03-reprint.pdf