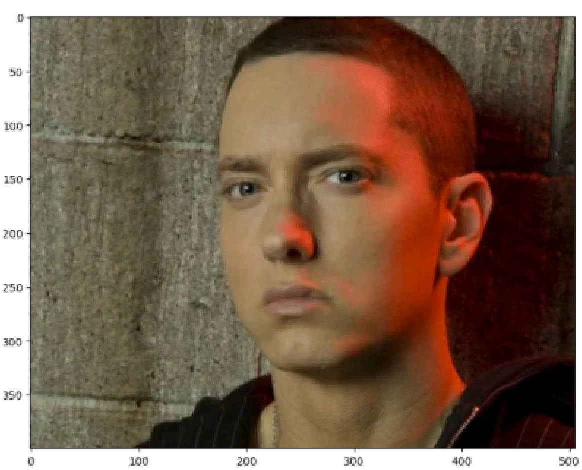
Chris Chen
Github:
https://github.com/compscichris/CSCI611_Summer25_Chris_Chen/tree/main/Assignment%203

Part 1: I was able to follow the skeleton code that guided me in using the VGG-19 CNN to apply the Image Style Transfer, detailed in the paper by Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. I was able to understand how to use the pytorch libraries to flatten the input and transpose to calculate the Gram-matrices used for texture feature extraction. My end result looks a little stranger than I anticipated, and I included the first result so far.

The **first run** seemed a little fishy, so I verified my code. I fed the image into the gram matrix, which led to unexpected results. The color was wrong. I changed that, and fed the individual features into image, and it gave me just the original image. That was because the gram matrix was the improper shape, I flattened it to batch, feature*spacials, instead of feature, spacials. After changing it, I got the results of the **second run**.
***second run** had total loss of

```
24.14239501953125
12.477807998657227
8.492374420166016
6.720818519592285
5.793830871582031
```



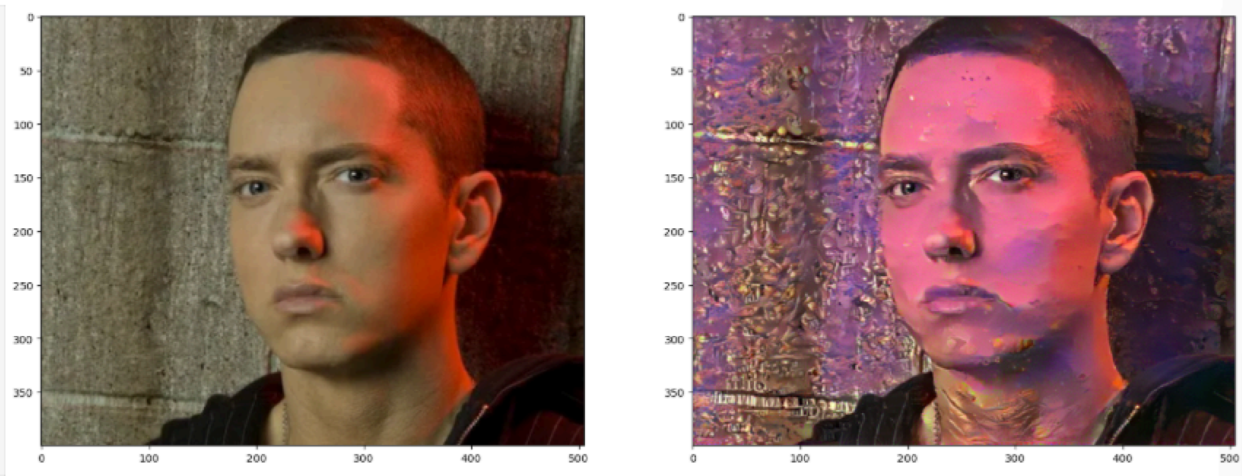
Part 2
My first modification was changing the steps.
The base was 2000, and I tested 3000, 4000, and 1000.

stepEdit1 was 3000 step, and yielded a total loss of

```
24.140098571777344,
12.473718643188477,
```

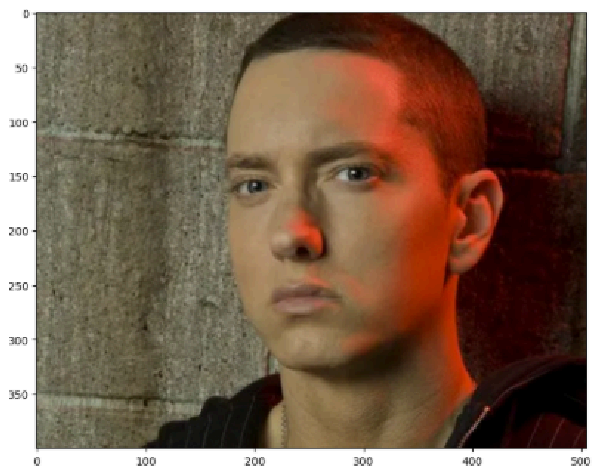
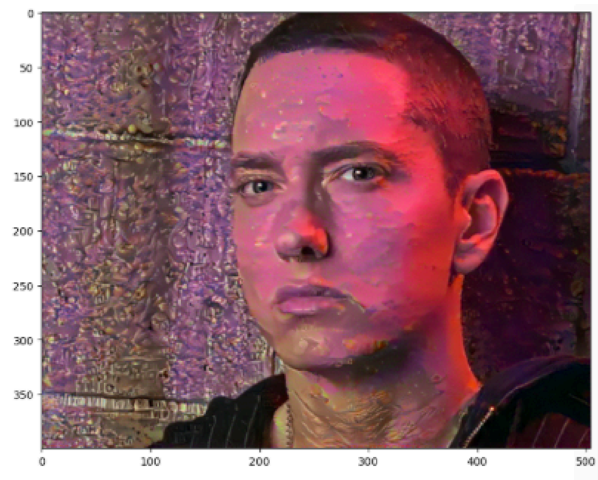
8.490579605102539,
6.719324111938477,
5.794463157653809,
5.246639251708984,
4.88814640045166

```python
# display content and final, target image
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
ax1.imshow(im_convert(content))
ax2.imshow(im_convert(target))
plt.show()
```



stepEdit2 had a mistake, but I tested 4000 without refreshing.
3.621861696243286,
3.6169984340667725,
3.613125801086426,
3.6072707176208496,
3.6004316806793213,
3.5945520401000977,
3.5872647762298584,
3.5828256607055664,
3.5791311264038086,
3.5735409259796143 were the loss values, but it seems the more you steps passed through, the more intermediate images are generated as the model learns and reduce the total loss.

stepEdit3 was 1000 step, and yielded a total loss of
24.142976760864258,
12.477656364440918

What I noticed was that more steps led to less total loss, and much more intermediate images.

My second modification was changing the alpha and beta values.

1e6 base total loss =

```
24.14239501953125
12.477807998657227
8.492374420166016
6.720818519592285
5.793830871582031
```

The base was 1e6 for beta, and I tested 1e5 and 1e7.

```
betaEdit1 total loss 1e5
24.140886306762695,
12.47514533996582,
8.493656158447266,
6.720850944519043,
5.795681476593018

betaEdit2 total loss 1e7
24.139070510864258
12.482877731323242
8.491659164428711
6.7243804931640625
5.800724983215332
```

$$\mathcal{L}_{\text{total}}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{\text{content}}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{\text{style}}(\vec{a}, \vec{x})$$

By changing the value of beta, I expect that the total loss would increase with a higher coefficient, and decrease with a lower one. But the results appeared inconsistent, so I cannot really draw a conclusion from the results, on what the exact impact of changing beta and alpha are. It may be affected by the weights.