Name: Batuhan

Surname: Yalçın

ID: 64274

COMP527/ELEC519, Spring 2023- Programming Assignment 5: Text-driven Image Manipulation

Table of Contents

Part 1 Getting Started with StyleGAN2

Truncation Parameter:

Truncation 0.1:

Truncation 0.2:

Truncation 0.3:

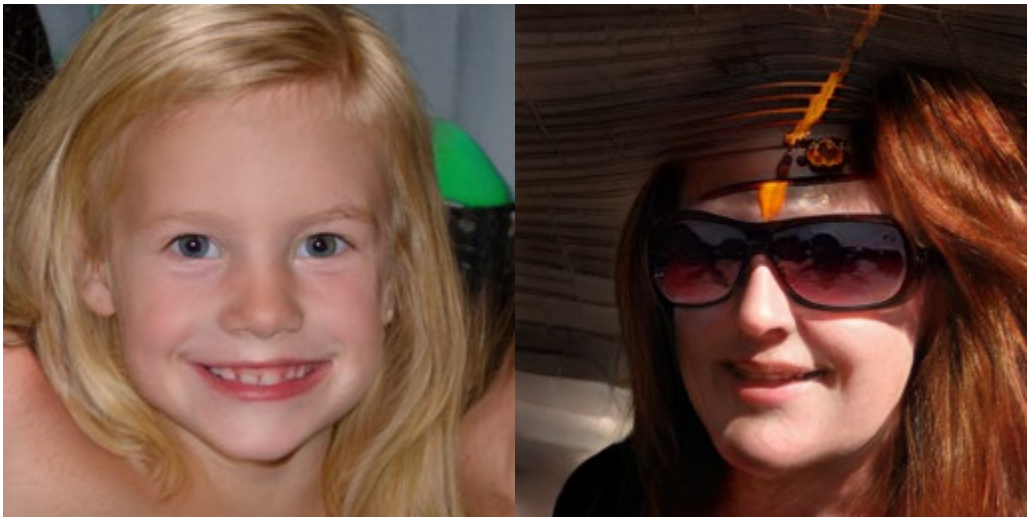
Truncation 0.4:

Truncation 0.5:

Truncation 0.7:

Truncation 0.9:

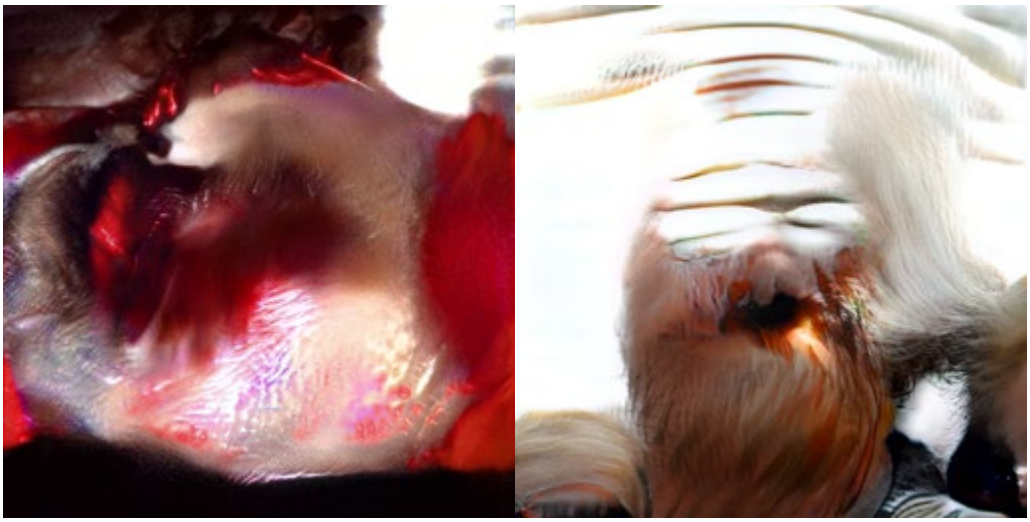
Truncation 1.1:



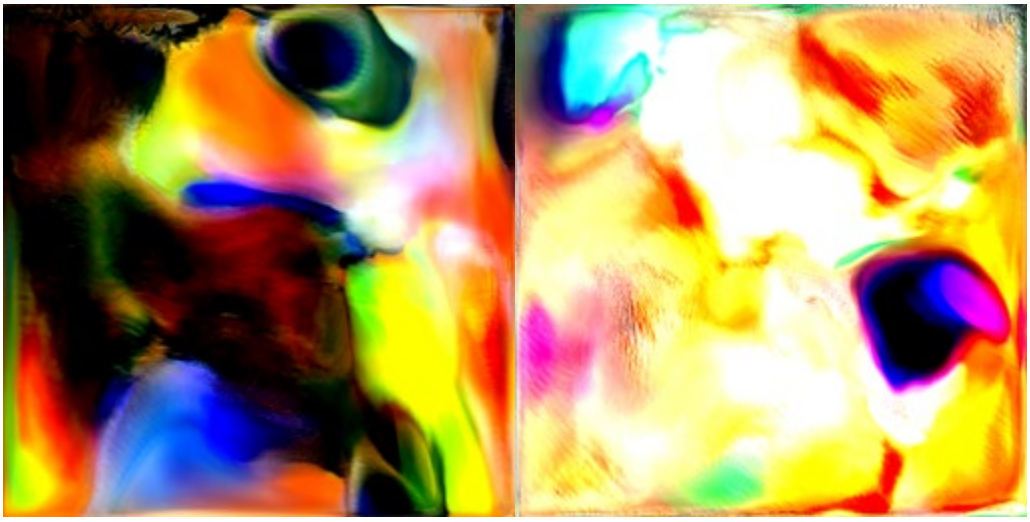
Truncation 1.3:

Truncation 1.5:

Truncation 2:



Truncation 5:



Discussion

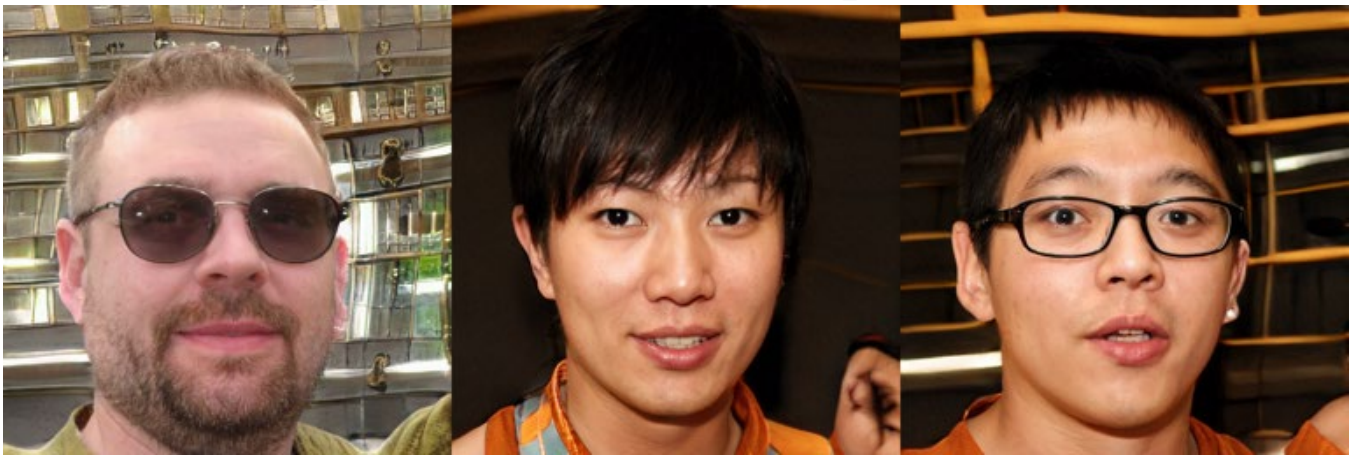
In lower truncation factors, the exploration in the latent space is reduced, resulting in more realistic images. When increasing the truncation factor to 0.7, the images remain realistic, but more diverse features are explored and added to the image. However, when using truncation factors beyond 0.7, such as 0.9, unrealistic alignments and the loss of specific features, like the hat-wearing girl in the right image, become more apparent.

As the truncation factor exceeds 1.0, the exploration between images becomes more pronounced, but the overall realism is significantly compromised. Even at higher values like 2.0 and 5.0, the generated images become highly imaginative but lack coherence and meaningful representations. Nevertheless, I find them intriguing and could consider creating NFTs to sell in NFT markets, appreciating their artistic value. 😊

Style Mixing:

First 1 style vector from w_1 and the remaining 17 style vectors from w_2:

First 4 style vectors from w_1 and the remaining 14 style vectors from w_2



First 8 style vectors from w_1 and the remaining 10 style vectors from w_2

First 12 style vectors from w_1 and the remaining 6 style vectors from w_2

First 1 style vector from w_2 and the remaining 17 style vectors from w_1

First 4 style vectors from w_2 and the remaining 14 style vectors from w_1

First 8 style vectors from w_2 and the remaining 10 style vectors from w_1

First 12 style vectors from w_2 and the remaining 6 style vectors from w_1

Discussion:

Each style vector's dimension in StyleGAN regulates a certain semantic feature or visual aspect of the synthesized image. For instance, dimensions can regulate characteristics like age, gender, position, illumination, or skin tone. It is clear that the initial stages record low-level characteristics like edges and textures, but the latter stages gather high-level semantic data like object shapes and structures.

A brief comparison follows:

When mixing latent at different phases, the results show changes in both general characteristics and precise details. The synthesized visuals transition toward the look of the second sample and integrate increasingly high-level semantic features as the mixing moves toward its conclusion.The earliest phases of mixing tend to produce synthesized images that are more like the first sample than the second, especially in terms of overall form and structure. The effect of the second sample becomes increasingly noticeable as the mixing moves into its final stages, causing a progressive shift towards its qualities.

In terms of overall form and structure, mixing at the earlier stages frequently results in images that resemble the first sample. As the mixing progresses, the second sample's effect becomes more noticeable, which causes a shift toward its attributes.

Specific semantic characteristics are within the control of each latent space dimension. In the initial phases, low-level details are captured, such as edges, textures, eyes, and ears. Higher-level semantic aspects, such as object shapes, global organization, and overall appearance, are captured in later phases.

Examples of Semantic Feature Control:

Ears and Eyes Transition: From sample 1 to sample 2, the ears and eyes are gradually altered by mixing from the First 1 style vector from w2 to the First 12 style vectors from w2. This demonstrates how earlier phases influenced the ability to capture subtle facial feature information.

General form Control: Even though more of the latent space belongs to w2, the general form of the head is similar to that of sample 1 in both the first four style vectors from w1 and the remaining 14 style vectors from w2. This exemplifies the role of initial latent vectors in shaping and influencing the overall structure.

Part 2 Editing Images with StyleCLIP-LO)

CLIP Loss:

It was straightforward for me to follow the instructions, so here is my code:

```
        # Pass the image through the upsample and average pool layers
        # Pass the image & text through the CLIP model, scale the output by dividing by 100
        # Model output is the CLIP similarity, subtracting the similarity from 1 will give us the distance

        pass1 = self.avg_pool(self.upsample(image))
        pass2 = (self.model(pass1, text)[0])/100
        distance = 1 - pass2

        ### YOUR CODE HERE
        return distance
```

ID Loss:

The ID (identity) loss is used to preserve the similarity between the synthesized images and the original image. It helps in ensuring that the generated image retains the characteristics and features of the original image, acting as a form of image preservation.

L2 Loss:

I simply calculated the difference between the two latents, squared the result, and summed all the entries in the resulting vector to obtain the loss value.

Latent Optimization Procedure:

In the first implementation, I generated the original image from w_init using the provided example in the cell below. By setting additional parameters such as return_latents=false and randomized_noise=false, I ensured that the final output is the resulting image rather than the intermediate latents.

```
# Generate the original image from w_init, you will use this for the ID loss
with torch.no_grad():
    #### YOUR CODE HERE ####
    original_image,a = G([w_init], input_is_latent=True, return_latents=False, randomize_noise=False)
```

Next, I initialized the loss functions by simply calling them.

```
# Initialize your loss functions:

#### YOUR CODE HERE ####
l2_loss = L2Loss()
id_loss = IDLoss()
clip_loss = CLIPLoss()
```

Then, as I mentioned earlier, I cloned the initial latent and set its requires_grad field to true. This allows for backpropagation during the optimization process.

```
# Create a clone of the initial latent, then set the requires grad field to True

#### YOUR CODE HERE ####
w = w_init.clone().requires_grad_(True)
```

Following that, I initialized the optimizer with Adam and set the learning rate to 0.1. This modification ensures that the learning rate remains fixed during the optimizer loops.

```
# Initialize the optimizer, you should use Adam with a learning rate of 0.1:

#### YOUR CODE HERE ####
optimizer = torch.optim.Adam([w], lr=0.1)
```

During the optimization loop, in each iteration, I generated a new generated image from the updated latent. This time, I did not use torch.no_grad because I needed to update each image during the loop iteration and adjust the lighting. The explanation of the parameters is described above:

```
#### YOUR CODE HERE ####
generated_image,b = G([w], input_is_latent=True, return_latents=False, randomize_noise=False)
```

To compute the total loss, I calculated the ID loss between the original image and the generated image and scaled it by 0.005 as described. I also calculated the L2 loss between the initial latent and the optimized latent and scaled it by 0.005. Additionally, I calculated the clip loss between the generated

image and the text inputs. Finally, I summed all the losses together.

```
# Calculate and sum all the 3 losses (don't forget the set the weights of the losses)

#### YOUR CODE HERE ####
loss_id = 0.005 * id_loss(original_image, generated_image)
loss_l2 = 0.008 * l2_loss(w_init, w)
loss_clip = clip_loss(generated_image, text_inputs)
loss = loss_l2 + loss_id + loss_clip
```

Using the optimizer I created earlier, I called the zero_grad function, which zeroes out the gradients of the parameters. This step is necessary before calculating the gradients and performing a backward pass. Then, I called the loss.backward function to perform automatic differentiation and calculate the gradients of the loss with respect to the parameters that have requires_grad=True. Finally, I called the optimizer.step function to update the parameters based on the computed gradients and the specified optimization algorithm.

```
# Backward the loss and update the parameters

#### YOUR CODE HERE ####
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Finally, I returned the final image. However, before doing so, since in the last iteration the loss was again backwarded and the parameters were updated, I regenerated the final image and then returned it.

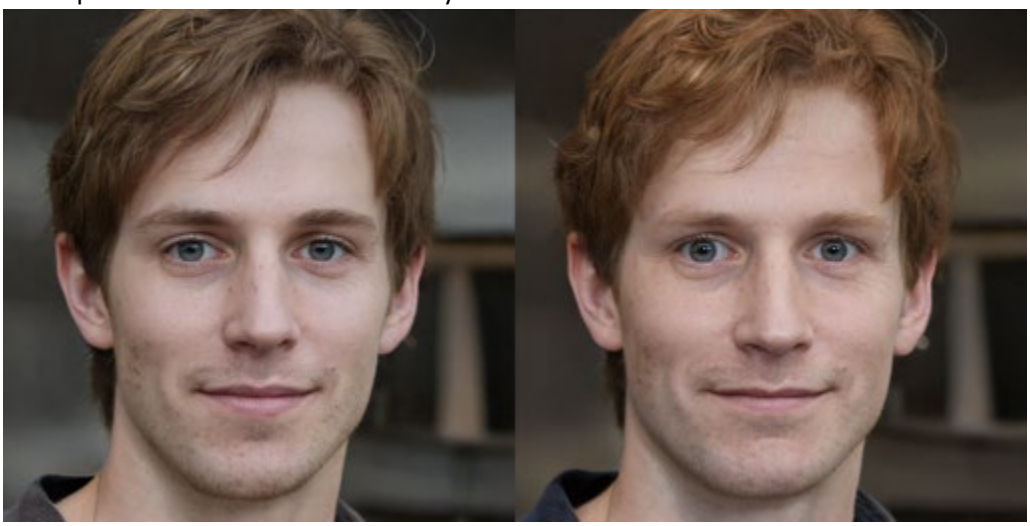
```
#### YOUR CODE HERE ####
final_image,c = G([w], input_is_latent=True, return_latents=False, randomize_noise=False)
return final_image
```

Final images:

Description: A man with a beard:

Description: A man with a scary face.

Description: A man who loses in every election.



Description: Kemal Kilicdaroglu.

Approved

Description: A man with a beard:

Part 3 Coarse-Medium-Fine Latent Analysis)

In this part, as explained in the explanation, I first clone the chosen chunk and set the requires_grad field of only the corresponding chunk. Then, I pass the optimizer only that chunk.

```
#### YOUR CODE HERE ####
w_coarse = w_init[:,:4,:]
w_medium = w_init[:,4:8,:]
w_fine = w_init[:,8:,:]
if chunk=="coarse":
  w = w_coarse.clone().requires_grad_(True)
elif chunk=="medium":
  w = w_medium.clone().requires_grad_(True)
else:
  w = w_fine.clone().requires_grad_(True)
```

```
# Initialize the optimizer, you should use Adam with a learning rate of 0.1:

#### YOUR CODE HERE ####
optimizer = torch.optim.Adam([w], lr=0.1)
```

To concatenate the other parts of the latent (chunks) and generate a new image in the loop, I create a function called get_w_concat. This function uses the torch.cat function to concatenate all three chunks

and generate a new image..

```
def get_w_concat(w1,w2,w3):
  w_concat= torch.concat(((torch.concat((w1, w2), dim = 1)), w3), dim = 1)
  return w_concat
```

Using this newly concatenated and updated latent, I generate a new image. The rest of the process remains the same as in the previous part.

```
#### YOUR CODE HERE ####
if chunk=="coarse":
 w_concat=get_w_concat(w,w_medium,w_fine)
elif chunk=="medium":
 w_concat=get_w_concat(w_coarse,w,w_fine)
else:
 w_concat=get_w_concat(w_coarse,w_medium,w)

generated_image,b = G([w_concat], input_is_latent=True, return_latents=False, randomize_noise=False)
```

Here are the results:

Coarse:

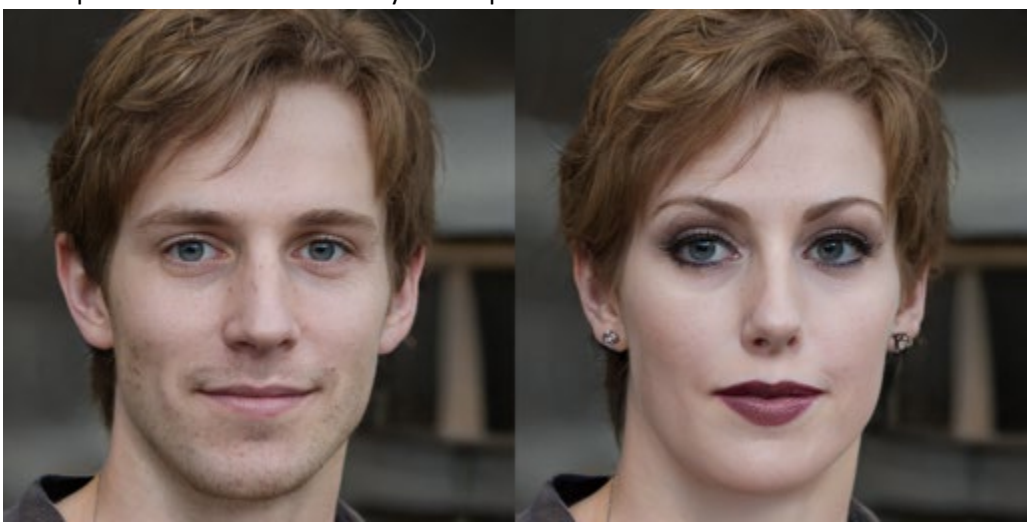
Description: A man with a beard:

Description: A woman with heavy makeup.

Medium:

Description: A man with a beard.

Description: A woman with heavy makeup.



Fine:

Description: A man with a beard.

Description: A woman with heavy makeup.

Discussion:

When controlling the chunks separately instead of combining them, significant differences were observed in the resulting images. Each chunk had a distinct influence on the generated images, capturing different aspects of the desired features.

The coarse chunk primarily controlled the overall structure and rough features of the generated images. It captured global structures and basic shapes while making minimal changes to the original image. For example, when the description mentioned "A woman with heavy makeup," the gender remained the same, and only subtle makeup changes were observed.

The medium chunk had a stronger impact on the generated images. It controlled the intermediate level of details and key features. When the description included elements like gender or specific attributes, the medium chunk brought noticeable changes to match those features. For example, in the description "A woman with heavy makeup," the gender was changed, and the image received extensive makeup. Similar transformations were observed in images involving facial hair.

The fine chunk focused on fine-grained details and intricate features of the generated images. It preserved the key features while updating them according to the text input. In the case of the description "A woman with heavy makeup," the gender remained the same, but the man in the image had some makeup applied. Similarly, in the beard version, the man had a beard, although not as pronounced as in the medium chunk or when all chunks were updated together.

When all chunks were optimized together, the model balanced and combined the coarse, medium, and fine details. The synthesized images reflected a more holistic representation of the target description, incorporating the desired features at multiple levels of detail.

Overall, controlling the chunks separately provided insights into their individual contributions and the specific aspects they influenced in the generated images. The combined optimization of all chunks allowed for a more comprehensive synthesis, incorporating coarse, medium, and fine details simultaneously..

Bonus: Editing My Own Photo)

Me:

Transform:

target_description = "A smiling man"

target_description = "Iron Man"

target_description = "Robert Downey Jr"

target_description = "An Assasin in Assasins Creed"

target_description = "A Sith Lord"

target_description = "A Jedi"

Kemal Kilicdaroglu:

Transform:

target_description = "A man who won the presidential election."

Surprisingly noting change but result in real life different.

target_description = "Farmer Kemal, Rural Kemal."

target_description = "Recep Tayyip Erdogan."

Kemal Erdoğan:

My selection was competition:

target_description = "Farmer Kemal, Rural Kemal."