Assignment 4

Chih-Hao Tsai

November 2024

1 Problem 1: *Music Generation*

1.1 My model and dataset

My model consists of 2 GRU layers and a few Dense layers. I choose to put 2 GRU layers in my model because I got the best result from this structure after trying different number of GRU layers.

The input size of the training dataset is (50, 3), which has 50 data points and each data point has 3 features (pitch, step, duration).

Layer (type)	Output Shape	Param #	Connected to
input_layer_27 (InputLayer)	(None, 50, 3)	0	–
gru_51 (GRU)	(None, 50, 128)	51,072	input_layer_27[0…
gru_52 (GRU)	(None, 128)	99,072	gru_51[0][0]
dense_24 (Dense)	(None, 64)	8,256	gru_52[0][0]
duration (Dense)	(None, 1)	65	dense_24[0][0]
pitch (Dense)	(None, 128)	8,320	dense_24[0][0]
step (Dense)	(None, 1)	65	dense_24[0][0]

Total params: 166,850 (651.76 KB)

Trainable params: 166,850 (651.76 KB)

Non-trainable params: 0 (0.00 B)

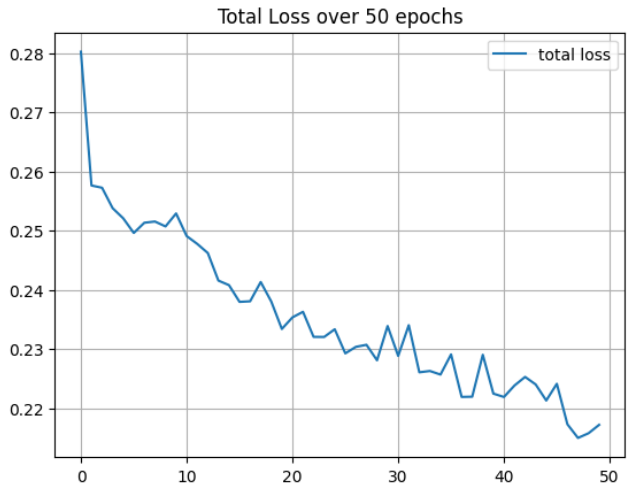


Figure 1: The left image shows the summary of my model (2 GRU layers with multiple dense layers), and the right image demonstrates the training loss over 50 epochs.

I created my dataset following the *Tensorflow Tutorial*. In the tutorial, they only use first 5 files of Maestro dataset (total of 1282 files) as their training dataset. I tried to use whole dataset as my training set, however, it's too time-consuming (it takes about 10 minutes only extracting notes from the dataset and has total of 7,126,318 notes). Therefore, I cut down the dataset to one third of the original dataset (50 files) as my training set, which has roughly 340,000 notes.

```
Total number of files: 1282
The number of files I use: 1282
Number of notes parsed: 7126318
```

```
Total number of files: 1282
The number of files I use: 50
Number of notes parsed: 338814
```

Figure 2: The image on the left is total files and total notes, the right image is the files and notes I used as my training set.

1.2 Parameters in generating notes

I set my parameters as the hours, minutes, seconds of current time. The relations between parameters are shown as following equations:

$$pitch = \frac{Preditcted\ Pitch}{hours},\ hours = \begin{cases} 1 & \text{if } hours = 0, \\ hours & \text{otherwise.} \end{cases}$$

$$duration = max(Predicted\ Duration \times random\_floating\_number(0, minutes),\ 0.01)$$

$$step = max(Predicted\ Step \times 0.01 \times random\_floating\_number(0, seconds),\ 0.1)$$

The *random_floating_number*() is a function that generates a random floating number between intended range, in my case, I use *random.uniform()* function in Python. For example, *duration* is the product of Predicted Duration (from model's prediction) and the random floating number between 0 and current minutes.

In generating duration and step, I set the lower bound for them (0.01 for duration and 0.1 for steps) so that the generated notes would not be too short . As for the reason why I multiply 0.01 when calculating step is simply because that makes the generated notes sounds better and cleaner.

With these parameters, the generated notes would be different if we run the code at different time of the day.

1.3 Generated Music

There are 240 notes that being generated with my own notes generating function. And the time I execute notes generating function was at 12:19:2 a.m. (Figure 3) Also, I summarized my music with a graph (Figure 4) illustrating generated pitches and their time stamps and duration, and a chart that shows the distribution of each generated parameters (pitches, steps, duration). ***Link to my generated music: GRU.mp3***

```
Current Time (HH:MM:SS): 0:19:2
```

Figure 3: The time I executed the notes generating function

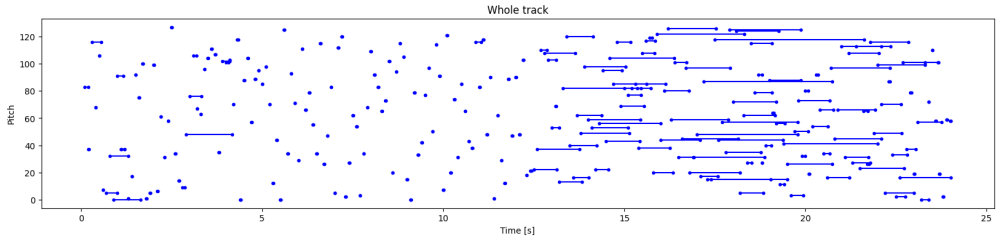


Figure 4: This graph shows all generated pitches of the whole track alone with their time stamp.

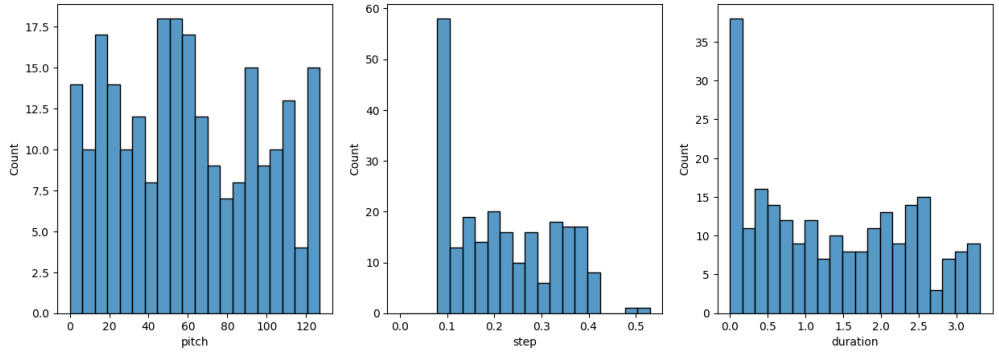


Figure 5: This chart demonstrates the distribution for generated pitch, step and duration.

2 Probelm 2: *My Own Positional Encoding*

In this problem, I use a 4096×3072 RGB image and process it into a 1200×1200 gray scale image as training image so that the dimension of MLP would not be too large to train.

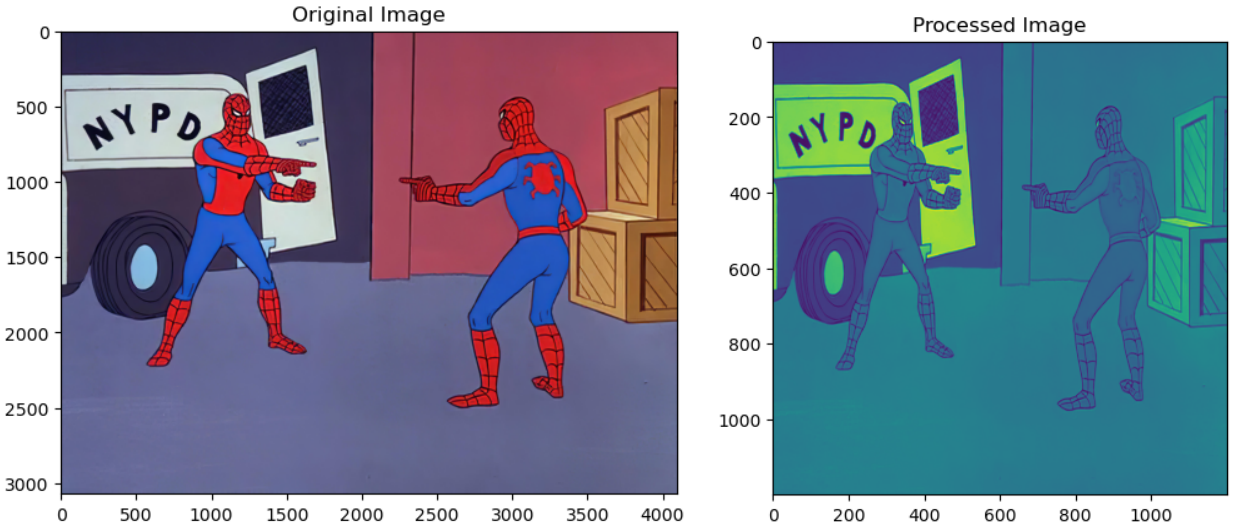


Figure 6: The left image is the original image (4096×3072, RGB image) that I got online, and the right image is the processed image I used as my training image.

2.1 Read SIREN paper

2.2 Reproduce SIREN architecture with higher resolution

In the tutorial, the team uses a 256×256 gray scale image as their training image. At first, I was confused about why does they use gray scale image instead of RGB image. Therefore, I tried to use my original RBG image and resize it into size of 256×256 as training image. To my surprise, add in 3 more RGB channels caused the memory usage increases exponentially. Even if I tried other ways to reduce the dimension of the RGB image (resizing it to 128×128, cropping some region, etc.), but those methods still didn't help.

Finally, I implement fitting higher resolution image on a MLP with training on a 1200×1200 gray scale image and adjusting batch size to 128 (it was only 1 in the tutorial). As can be seen in figure 7, the training loss starts to converge after about 200 steps of training loop.

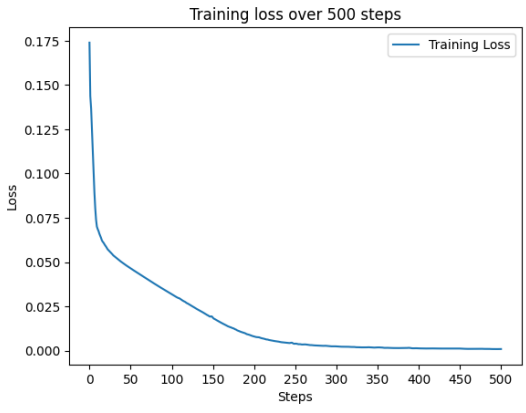


Figure 7: Training loss curve with provided sinusoid positional encoding.

In figure 8, we can see plots that present the generated images along with their gradient visualization and Laplacian visualization for each corresponding step. Also, these plots show 3 phases of image generating. In the beginning, step 0 is where the model starts from generating chaotic image. Then, in step 200, the contours of the original image start forming, this is also the step where the training loss curve starts converging. In step 500, the model is able to visualize some details of the original training image, such as the texture of the wooden boxes.

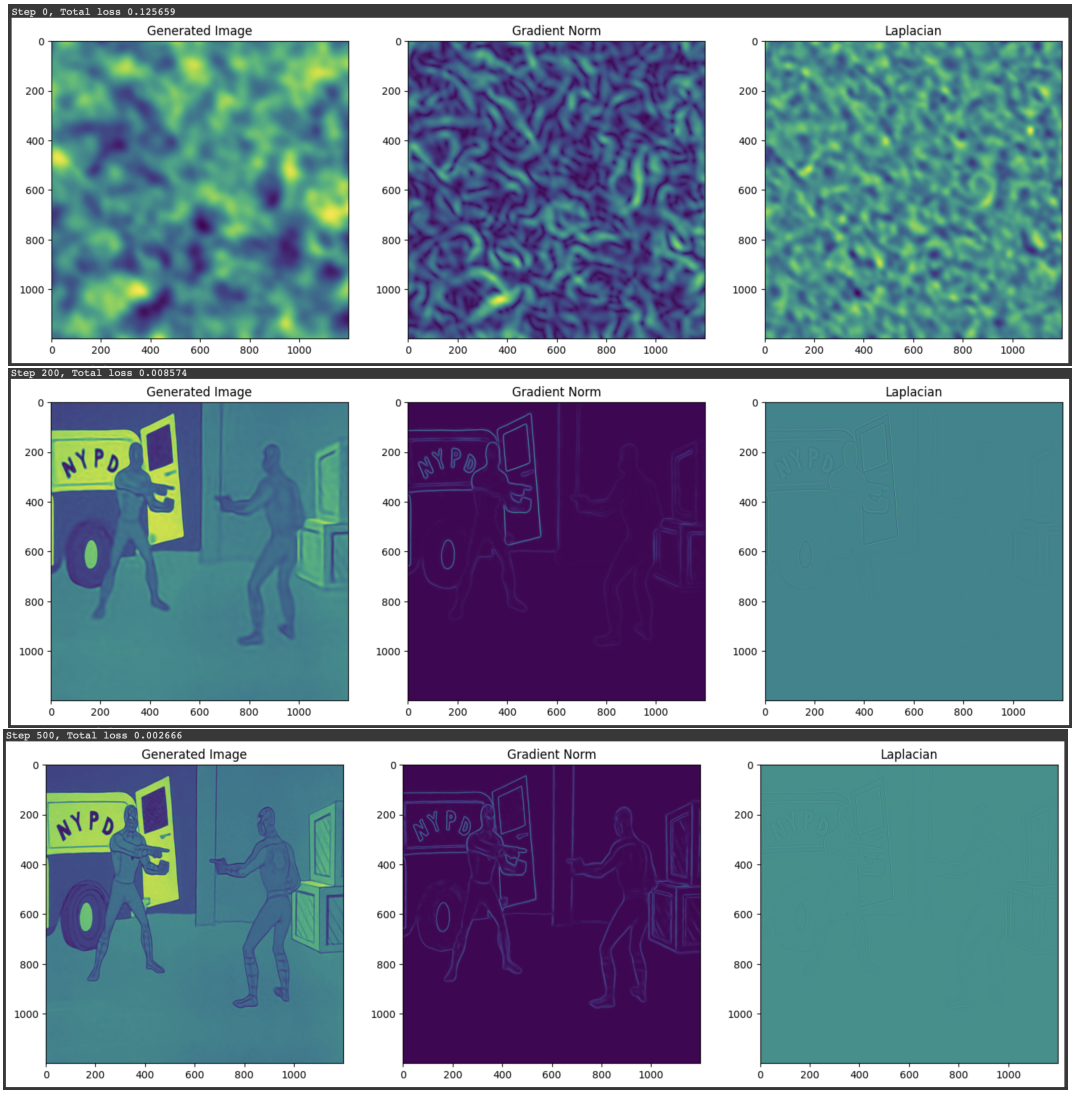


Figure 8: These plots demonstrate the generated images as well as their gradient and Laplacian derived maps with original SIREN.

2.3 Design my activation function

In this subproblem, I create another periodic function 1, which outperforms the original sinusoid function (the result is in section 2.4). The main idea of creating such function is to test whether the shape and the frequency of the activation function effects the performance of the model or not. Therefore, I create a Cosine composite function as my activation function (figure 9).

$$f(x) = 2cos(x) + cos(2.5x + 2) \tag{1}$$

Figure 9: My activation function in comparison to sin(x) function.

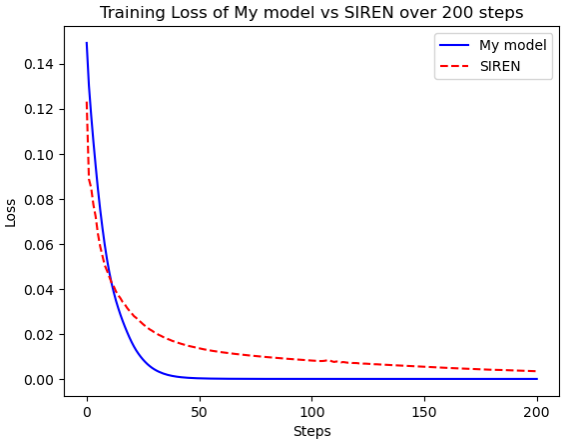


Figure 10: My model converge much faster than SIREN model.

2.4 The result of training on my activation function

Although I managed to train the SIREN model in section 2.1, it still took almost a day to train a 1200×1200 gray scale image for 500 steps. Thus, I trained 256×256 gray scale image beforehand to test if the model can get faster convergence with my activation function. To my surprise, my model outperform the original SIREN model (figure 10). Moreover, the loss of my model in 200th step even converges to less than 10^{-6} (figure 11).

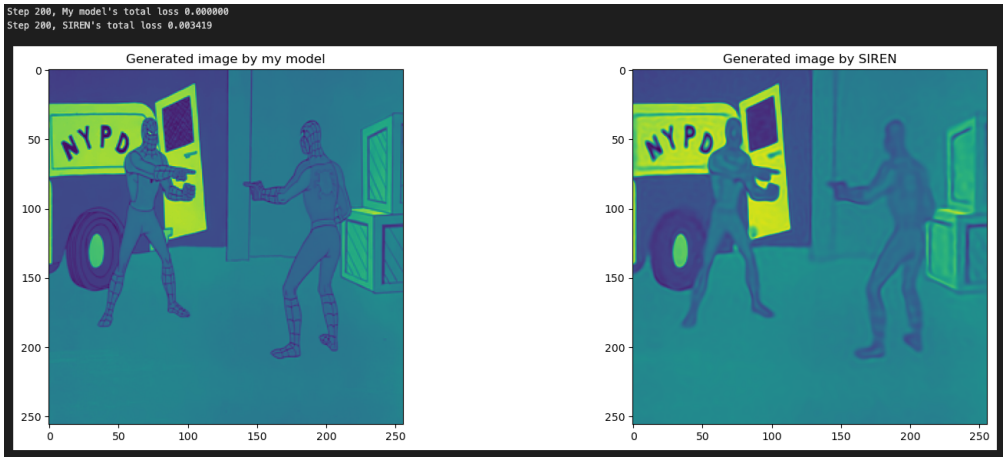


Figure 11: This plot shows that my model generates better image with fewer steps.

With the testing result above, I modify training steps on training my model to only 100 steps without changing any other parameters (batch size remain 128). As figure 12 demonstrates, training loss converges at around 45 steps of training session.

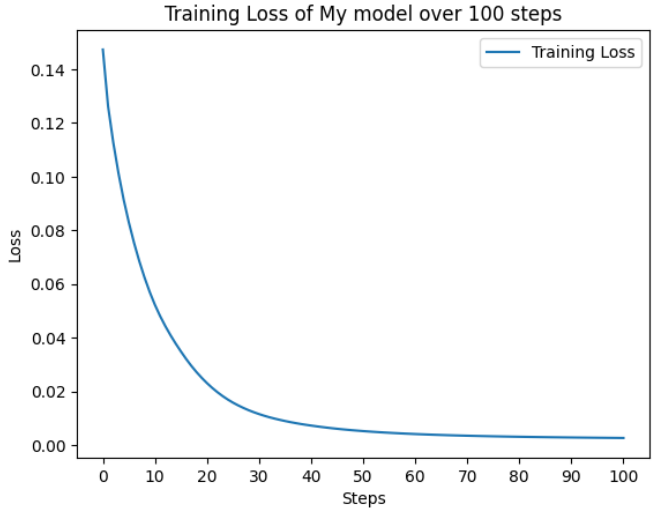


Figure 12: The training loss curve of training my model.

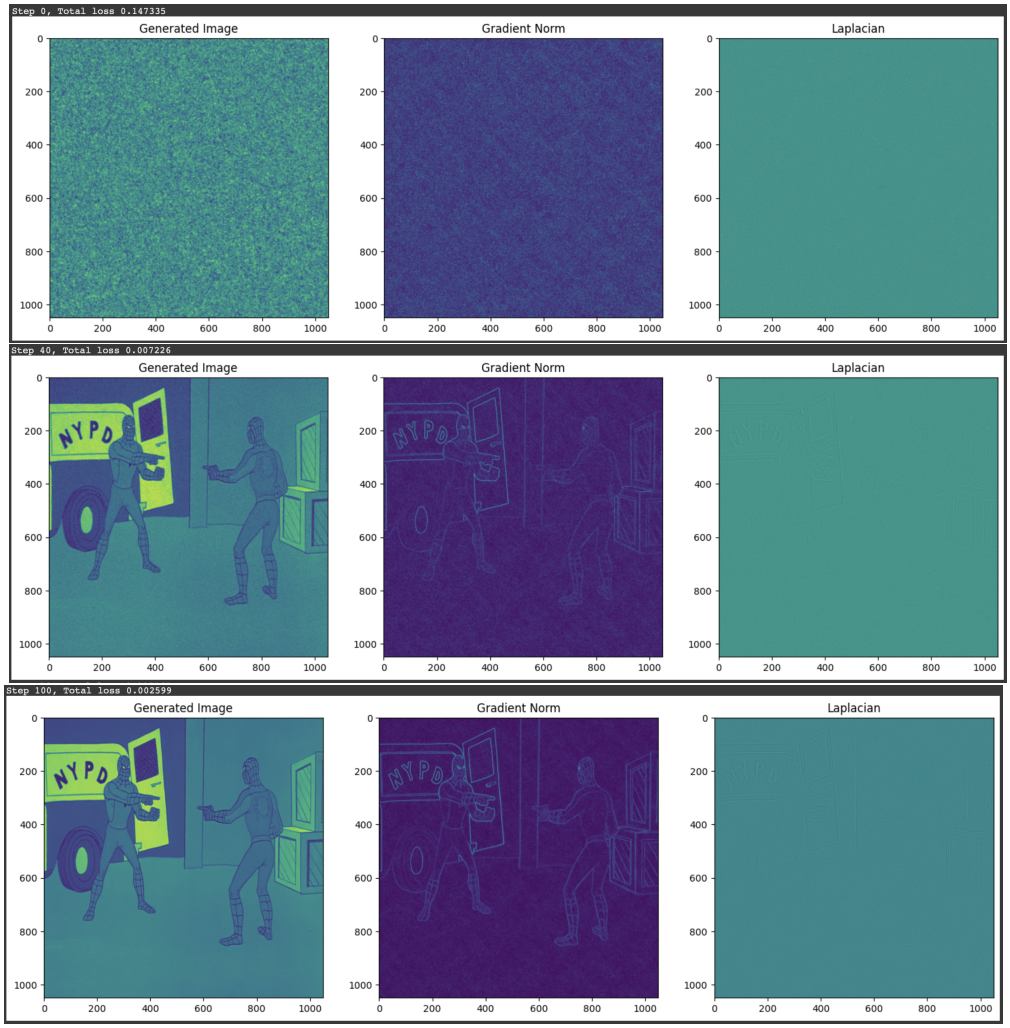


Figure 13: As can be seen in the plots in the middle (40th step), my model formed some details even in the early phase of generating image.

3 Problem 3: *Vision Transformer - Segmentation*

3.1 Run selfie image through a chosen face segmentation model

In this subproblem, I process my selfie to a 512×512, RGB image first. Then, I tried various models on Hugging Face Segmentation Models Library. The best image segmentation model for my selfie is *face-parsing* model, created by *jonathandinu*. This model is made for human-background segmentation, it can also mark clothing, facial features. The result is shown in figure 14.

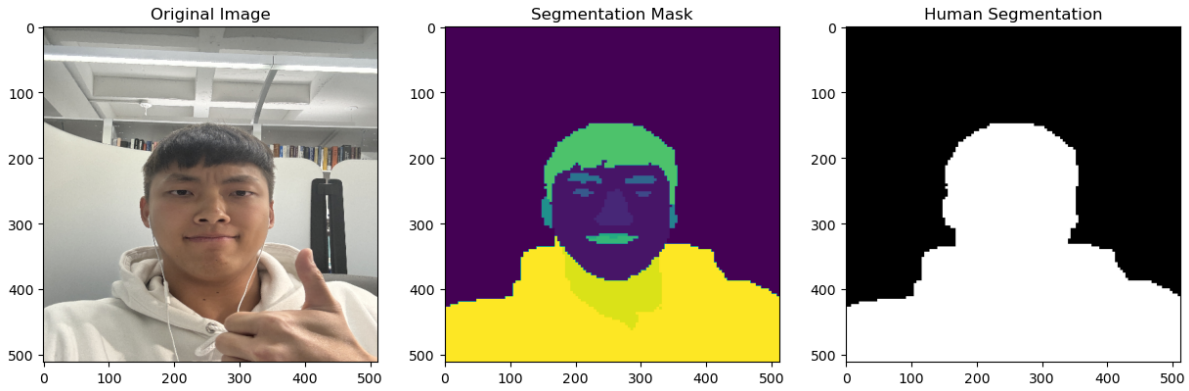


Figure 14: The result of implementing image segmentation.

3.2 Apply Gaussian blur to background

With the segmented mask, I implement Gaussian blur with $\sigma = 15$ to the background of the original image.

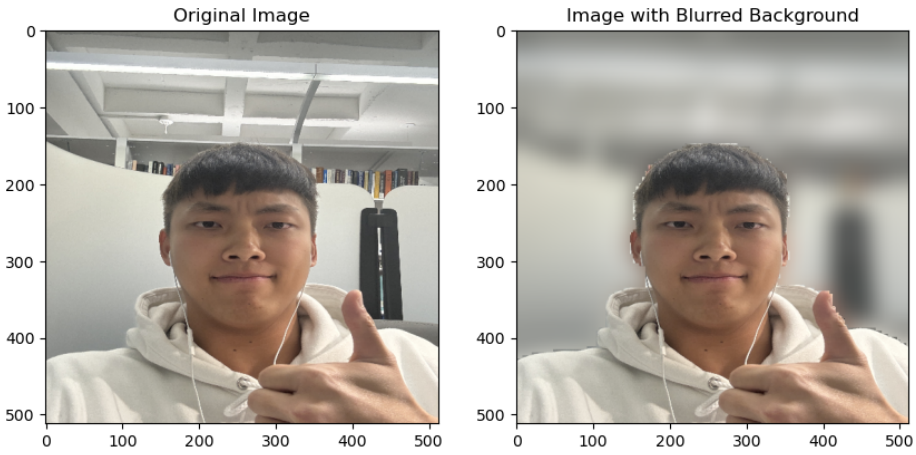


Figure 15: The result of applying Gaussian blur to the background.

3.3 Implement a chosen depth estimation model

I tried two depth estimation models in this problem and both of them worked out great, one is *depth-anything/Depth-Anything-V2-Small-hf* created by *Lihe Yang*, another is *Intel/dpt-large* created by *Intel*. In my provided images below and Google Colab notebook, I use the *Intel/dpt-large* as demonstration.

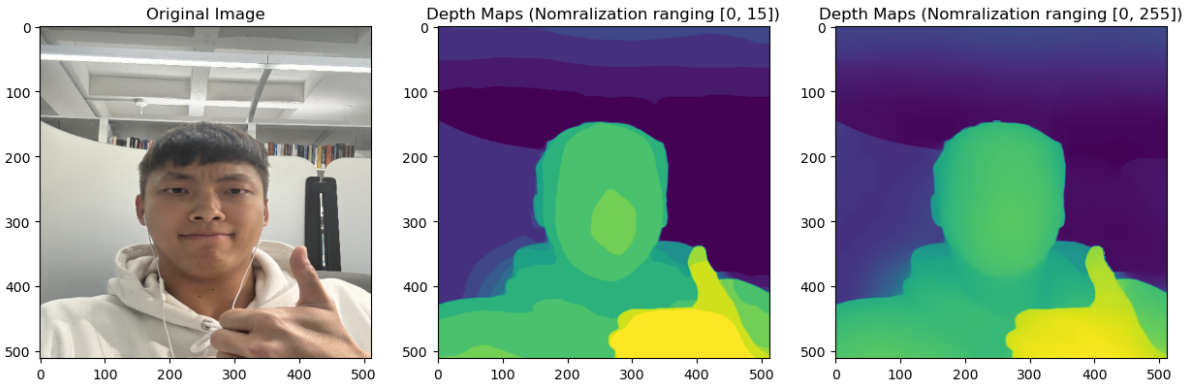


Figure 16: The result of applying depth estimation model based on different extend of normalization.

3.4 Implement Gaussian Blur according to different depth

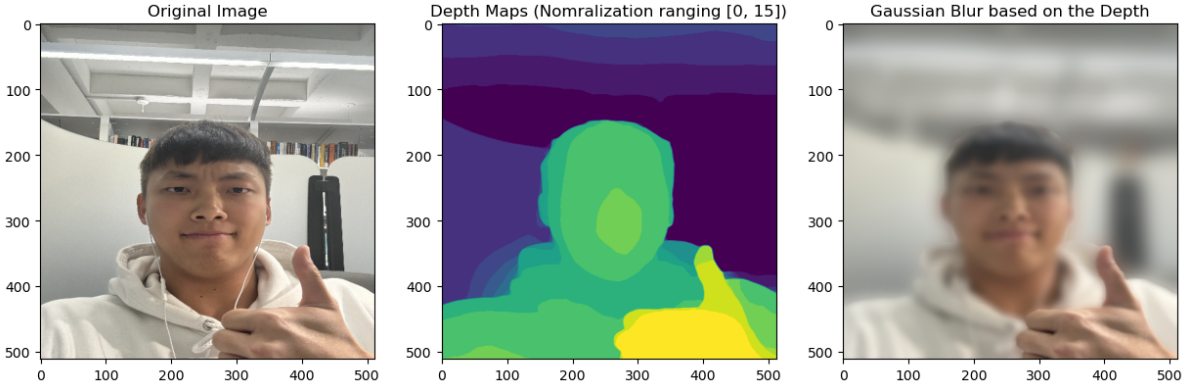


Figure 17: The result of implementing Gaussian blur based on different extend of depth.

3.5 Extra Credit

My Google Colab notebook Link: Link to my code.

4 Problem 4: *Generative AI*

4.1 DCGAN Tutorial

I followed all the instructions on understanding, building and implementing DCGAN network. When implementing DCGAN, I used the same parameters as the tutorial (batch size=64, learning rate=0.0002, beta=0.5, epochs=5), the images below are the result I got.

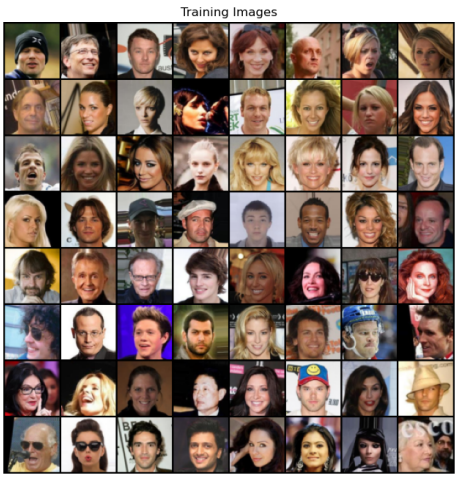


Figure 18: This is an example of training images I use for training.

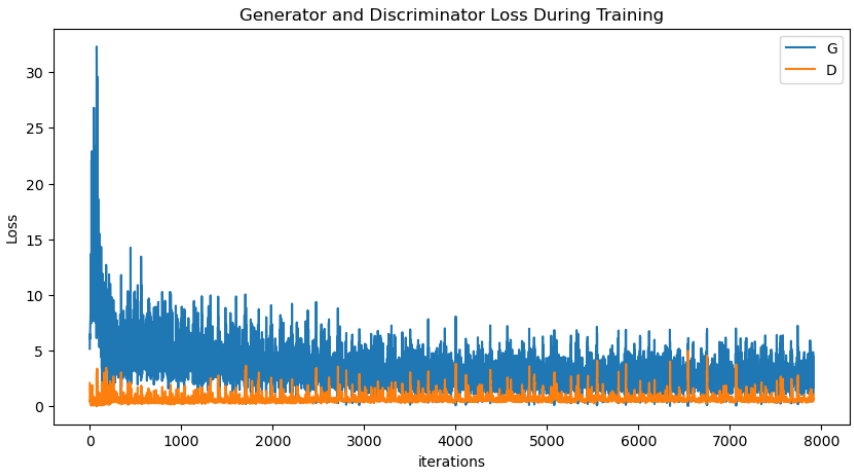


Figure 19: This graph shows the fluctuation of training loss of generator and discriminator during training session.

Figure 20: This is the comparison between real images and the generated images.

4.2 Create new colored squares dataset

I created an algorithm that can generated a 64×64 image with random sizes, colors, orientations squares in a black-background (figure 21). To test the network, I only generated 1000 training images at my first attempt.

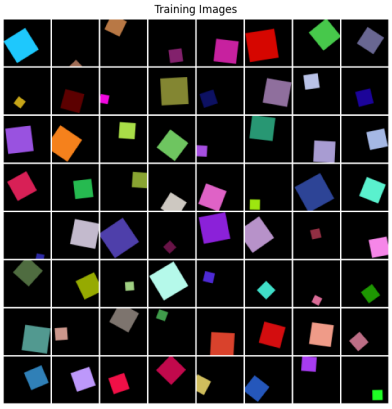


Figure 21: An example of generated images from the algorithm.

4.3 The result of training with colored squares dataset.

- **First Attempt**
  As I mentioned above, I only use 1000 images for the first training. Also, I didn't modify the parameters for training loop (batch size=64, learning rate=0.0002, beta=0.5, epochs=5). The performance of the network is quite bad due to lack of sampling data.

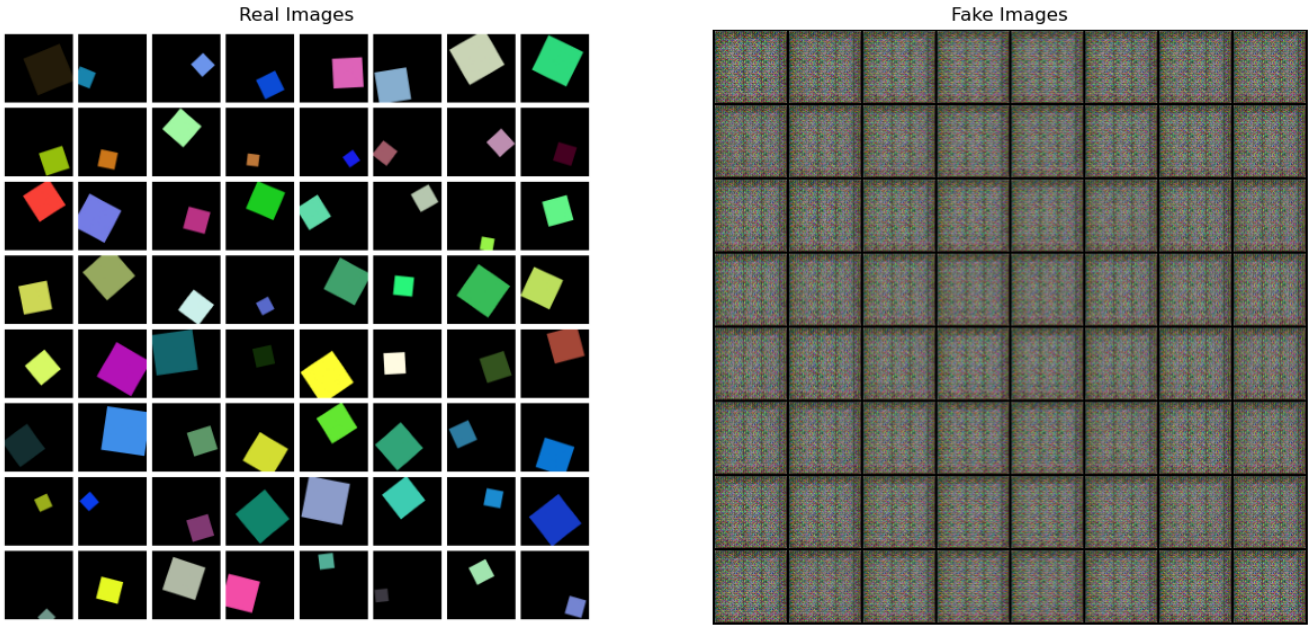


Figure 22: The result of the first training attempt. The network can barely form anything with the dataset.

- **Increase the size of dataset**
  Because of the bad result, I increased the size of the dataset to 150,000 images. Although a few of the generated images are still not clear enough, we still are able to tell most of the generated images are squares (figure 25).

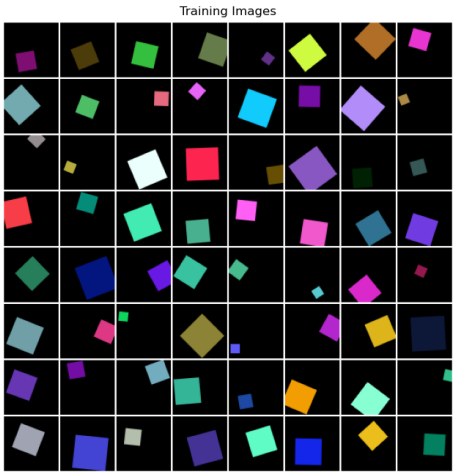


Figure 23: Some of the training image I use for training.

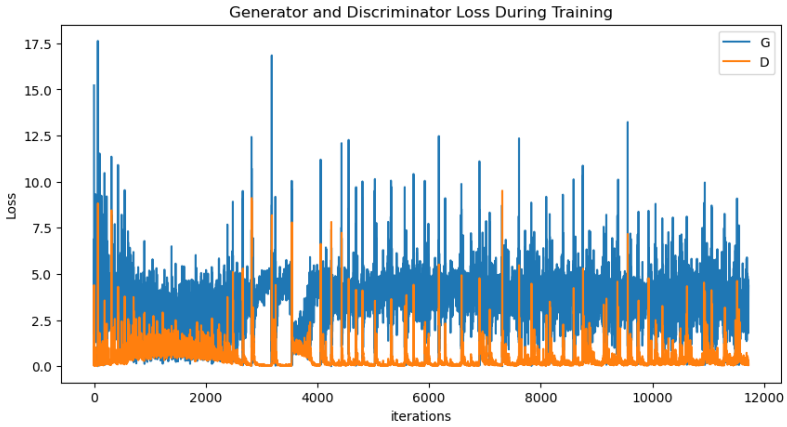
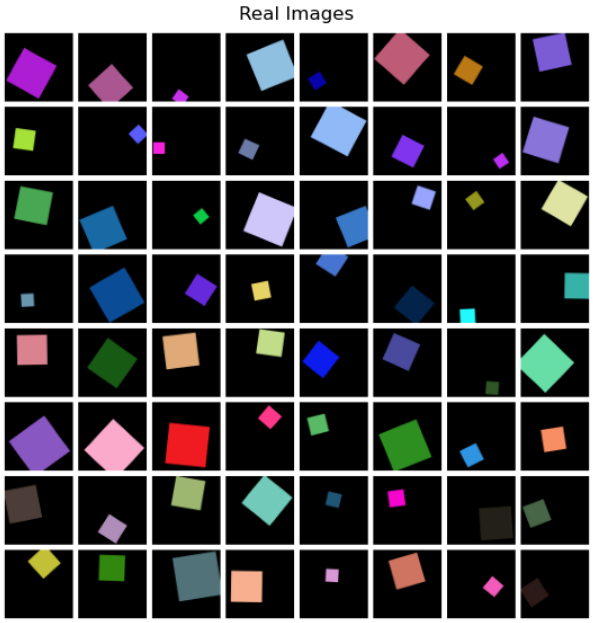


Figure 24: The training loss during training on a 150,000-images dataset.



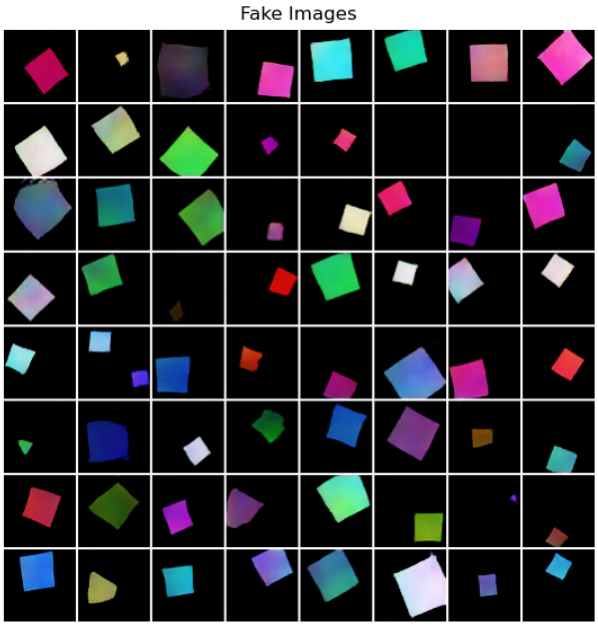


Figure 25: The result of training on 150,000 images dataset. Some generated images are clearly a square.

4.4 Collect new animal dataset

In this subsection, I downloaded a Cats faces 64x64 (For generative models) from Kaggle and perform image generation on the dataset. This dataset includes roughly 15,700 images of cats' faces from different breeds, different sizes, different angles.

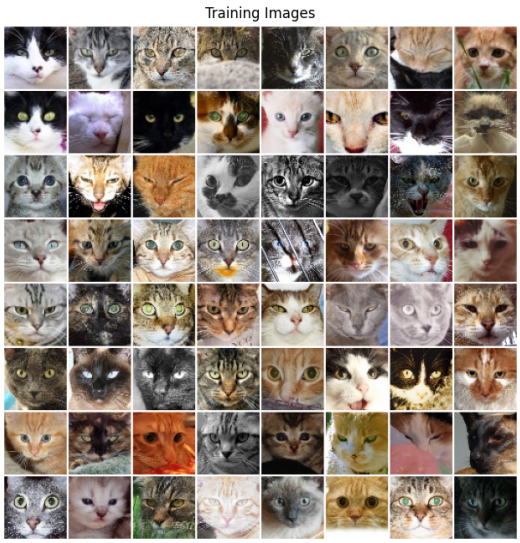


Figure 26: Some training images in the dataset.

4.5 Train DCGAN on animal dataset

- **First training attempt**
  In my first training, I use only 500 images for training dataset, batch size=64, learning rate=0.0002, beta=0.5, epochs=5. As my expectation, the result was not good. The network can only generated noise images.

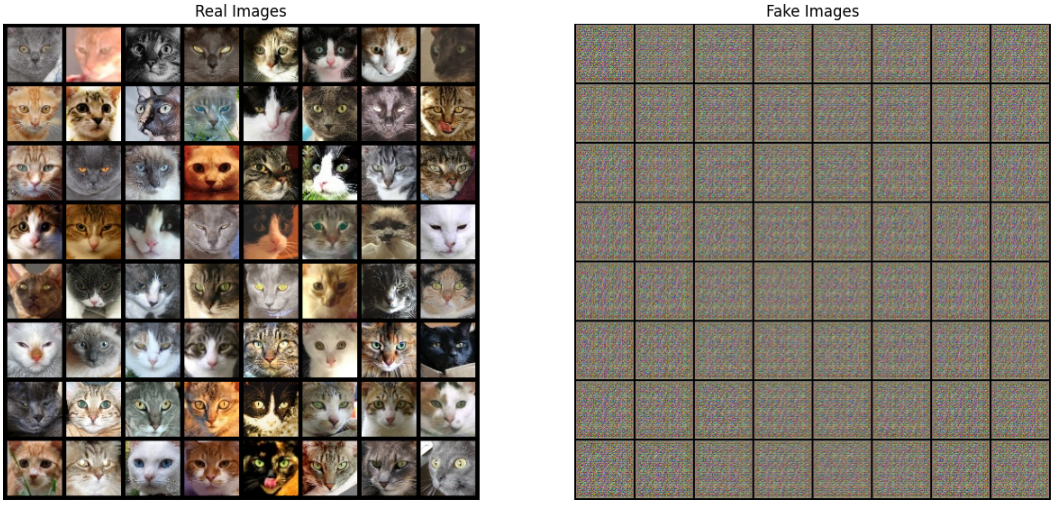


Figure 27: This image show the result of my first training result for new animal dataset. The network can only produce noise images with only 500 images dataset.

- **Increase the size of dataset and apply image augmentation**
  To get better performance, I tried various techniques. First, I increase the size of training dataset to 5,000 images. Then, I implemented image augmentation (randomly flipping, cropping, tilting the images) 28. Although the model was able to create some blurring facial features such as eyes, the overall performance is still not ideal (figure 29).

Figure 28: The training images after applying image augmentation.

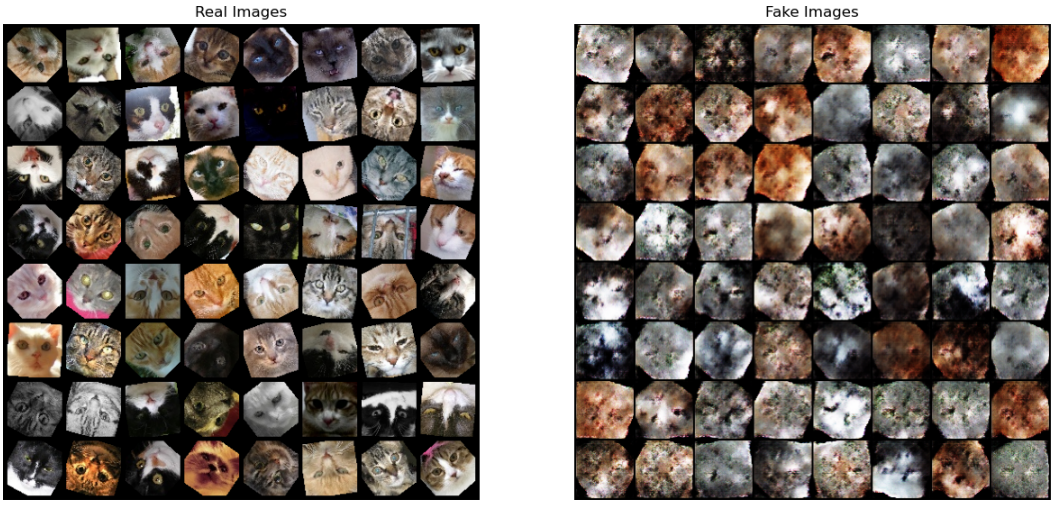


Figure 29: The result of increasing the size of dataset and performing image augmentation. The network can only create blurry images with a few facial features.

- **Full dataset with more iterations**
  To get more iterations, I set the batch size to 32 and training on full dataset (15,748 images). Then, I increased the number of epochs to 8. The performance improves a lot than the last attempt.

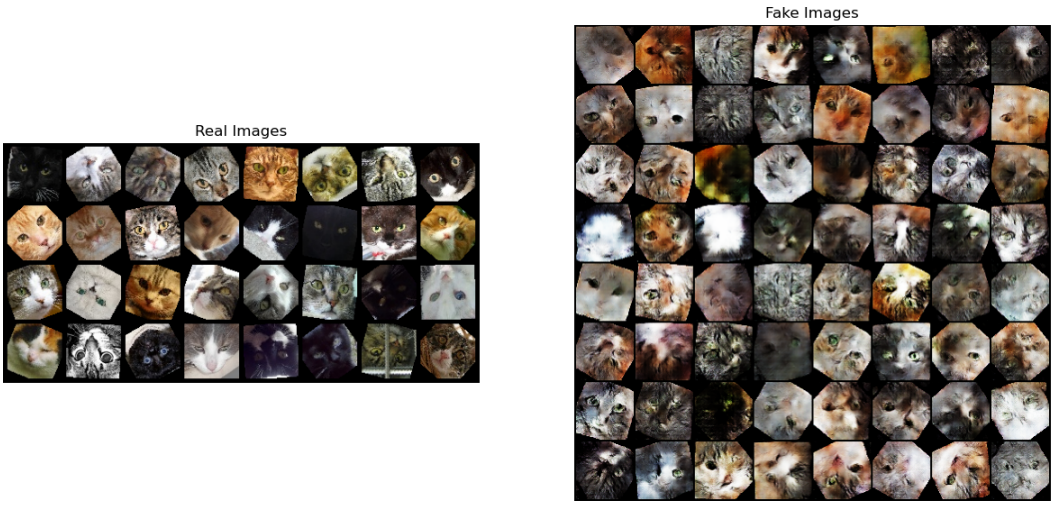


Figure 30: The result of increasing the number of iterations.

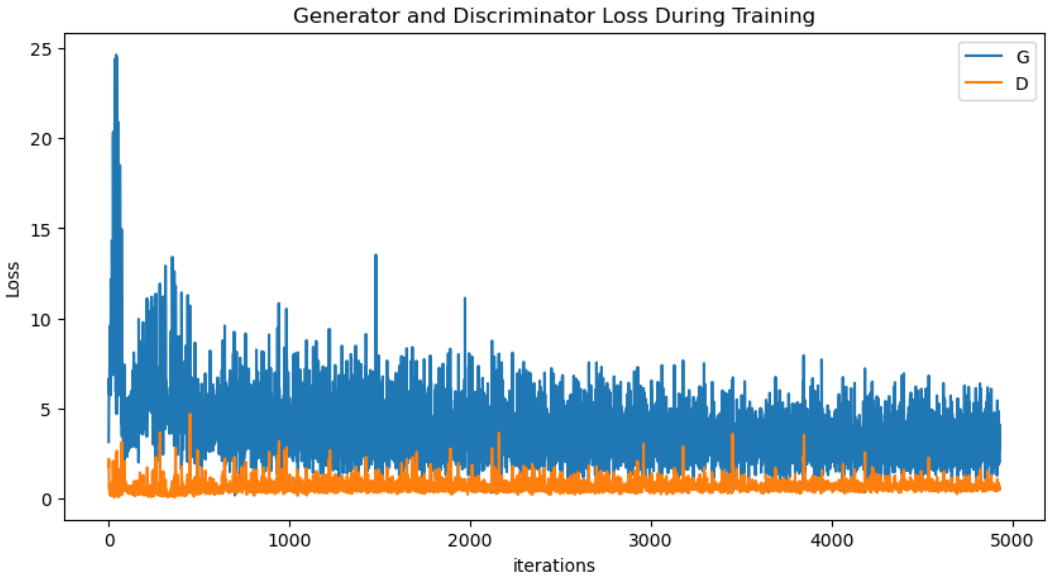


Figure 31: The training loss of increasing iterations.

- **Adjusting parameters**
  The result of increasing the number of iterations is fairly good. But I think there's something to improve. Therefore, I adjusted learning rate to 0.00025, which indicate increasing step size. Lastly, I deleted the image augmentation method, which I think might cause confusion to the model. The result is surprisingly good, the model is able to create some clear image of cat faces (figure 32).

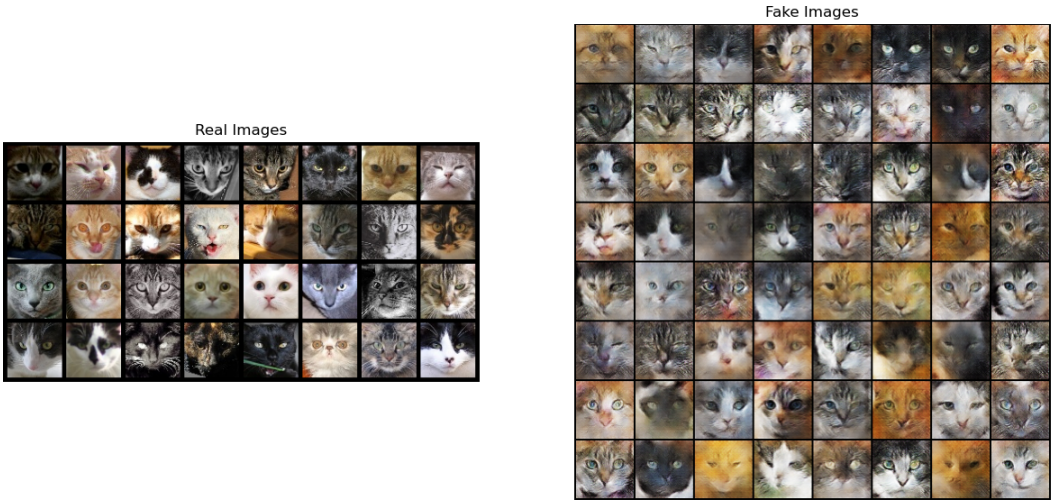


Figure 32: The generated images are fairly good.

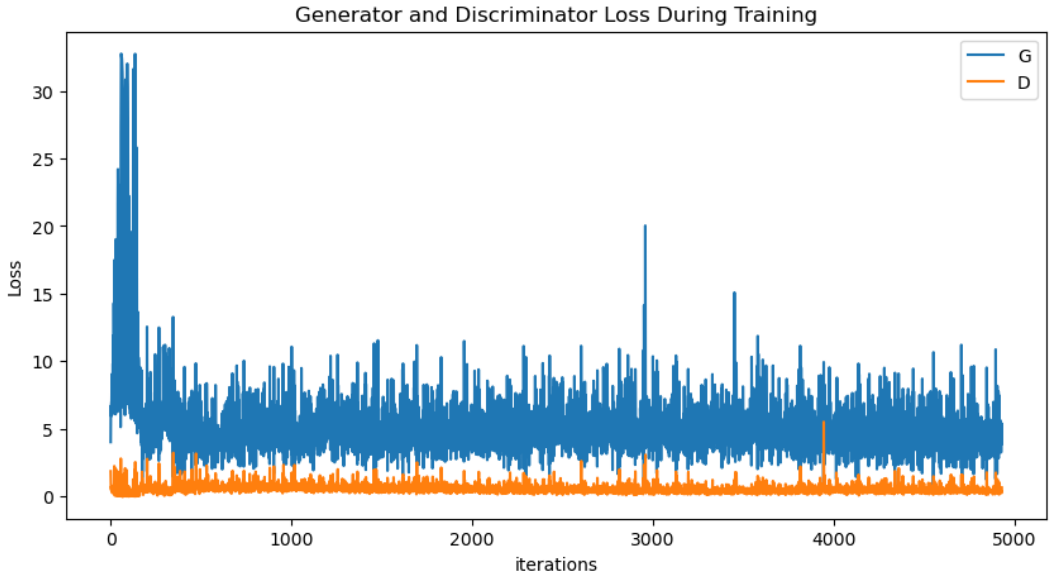


Figure 33: The training loss of my final attempt on training DCGAN.