Image-to-Image Translation with CycleGAN
Mason Wang and Qizheng Zhang

Introduction

In this project, we work on image-to-image translation with CycleGAN. In particular, we train CycleGAN to map real human faces to faces of anime characters, and vice versa. We use the selfie2anime dataset, which is available here: https://www.kaggle.com/arnaud58/selfie2anime. The training set is composed of 3,400 images of real human faces and 3,400 images of faces of anime characters that are unpaired. The test set is composed of 100 images of real human faces and 100 images of faces of anime characters that are unpaired.

We have implemented three versions of CycleGAN. Besides that, as a baseline, we train the official PyTorch code released by the authors of the paper. We do this to evaluate the quality of our own implementation.

The first version of our own implementation could be found in the notebook "Final Cycle GANs Pytorch v1.ipynb". The generator and discriminator architecture of this implementation largely follow those of the original paper "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks" [1]. We use least-squares loss for both the generator's adversarial loss and the discriminator's loss, use the same learning rate and optimizer, and opt to include the identity loss suggested in section 5.2. We keep an image cache of 25 instead of 50 images, and also have a different sampling procedure. The first version is based on references [3] and [4].

The second version improves upon the sampling procedure from the image cache from the first version. It adopts a different ratio of generator losses, suggested by [11]. The generator architecture changes slightly. Lastly, we implement a gradient penalty, following "Improved Training of Wasserstein GANs" [7], and "On the Effectiveness of Least Squares Generative Adversarial Networks". We reach the memory limits in Google Colab before we are able to finish an epoch of this model. The implementation can be found in the notebook "Final Cycle GANs Pytorch v2.ipynb".

The third version is identical to the second version, but without gradient penalty. This modification allows us to run the code and train the model without being bothered by memory issues. The implementation can be found in the notebook "Final Cycle GANs Pytorch v3.ipynb".

In this report, we will describe our model architecture, design choices, implementation strategies, and results & observations.

Baseline implementation

In order to obtain good baseline results for our implementation of Cycle-GANs, we train a Cycle-GANs network from https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix on the dataset of unpaired anime and real human faces. In this codebase, the implementation of the Cycle-GAN network is done for us by Jun-Yan Zhu*, Taesung Park*, Phillip Isola, Alexei A. Efros. (*equal contributions). This codebase corresponds to the paper "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks".

Version 1 Architecture

There are numerous differences between our implementation and the baseline model we trained using the code from "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks."

Loss function

While the original paper mentions the LSGAN loss, we did not train the baseline model using this flag. We did adopt least-squares loss in our model. Using least-squares loss alters our generator and discriminator objectives and is supposedly more "stable and resistant to mode collapse" (Maire, 2022). Below is the discriminator objective and the adversarial objective of our generators.

$$D^* = \arg\min_D \left[\mathbb{E}_{x \sim p_{\text{data}}} (D(x) - 1)^2 + \mathbb{E}_{z \sim p} (D(G(z)))^2 \right]$$

Push discrim. response on real data close to 1

Push response on generated data close to 0

$$G^* = \arg\min_G \mathbb{E}_{z \sim p} (D(G(z)) - 1)^2$$

Push response on generated data close to 1

In addition to the two adversarial losses, the two generators' loss also includes L1 cycle losses for each generator, weighted 5 times more than the adversarial loss, and identity losses for each generator, weighted 10 times more.

Discriminator

The architecture for our discriminator is based off of the architecture implemented in this tensorflow tutorial: https://machinelearningmastery.com/cyclegan-tutorial-with-keras/. This tensorflow implementation is based off of the architecture in the original paper.

The discriminator is composed of a stack of convolutional blocks that gradually increase the channel dimension while halving the spatial dimension at each step. In each convolutional block, we

1) Convolve the image with a kernel size of 2, stride of 2, and increase the number of channels
2) Apply instance normalization
3) Apply a LeakyReLU activation function (alpha = 0.2)

We increase the number of channels from 3 to 64 to 128 to 256 to 512. We convolve once more with a 512-channel to 512-channel layer, then perform instance normalization and apply a LeakyReLU activation. Then, we convolve down to a single channel. At this point, the spatial size of our layer is 16x16. The reason we convolve to an output layer of spatial dimension 16x16 instead of a single classification number is because a 16x16 spatial dimension may be more expressive, and could contain more information about where particular features occur on the image.

Instead of using Batch Normalization as the DCGAN paper does, we use instance normalization. This is recommended for tasks like style transfer because it erases style information [5].

Generator

The generator is an encoder, a stack of convolutional blocks with residual connections, and a decoder. It is based on reference [4]. The encoder is composed of three convolutional blocks. In each, we

1) Apply convolution (with preceding padding to match the kernel size), increasing the number of channels
2) Apply instance normalization
3) Apply a ReLU activation function

The channels in the generator increase from 3 to 64 to 128 to 256. The first convolution has a kernel size of 7, stride of 1, and we mirror-pad before it. The next two have kernel size 3, and a stride of 2.

Each of the 8 residually connected blocks in the middle does the following:

1) Convolves with kernel size 3, keeping the number of channels and spatial dimension constant
1) Instance normalizes
2) Applies ReLU activation
3) Convolves a second time, using a convolutional layer with the same structure as the first
4) Concatenates the current state of the signal to the input to the block
5) Applies ReLU activation

We can see that the end of each block is residually connected to the output from the previous block or encoder.

Lastly, in the decoder we have two convolutional blocks, each of which does the following:

1) Applies a transpose convolution with kernel size 3, doubling the number of channels
2) Applies pixel shuffling, doubling each spatial dimension and decreasing the number of channels by 4.
3) Applies instance normalization.
4) Applies ReLU

Then, we apply a final fractionally-strided convolution which mirrors the first convolution in the decoder, with a kernel size of 7, that produces an image output. We saturate this with a hyperbolic tangent activation.

Pixel shuffling was inspired by reference [4]. It is a technique applied post-transpose-convolution that condenses channels by replacing each spatial pixel in the input with shuffled versions of four pixels from different channels in the same spatial location.

Training design choices

Generative Adversarial Networks are notoriously difficult to train. In order to improve stability, our training procedure stores a cache of 25 fake anime faces and 25 fake real human faces that have been previously generated by our two generators. For most iterations, the discriminator takes in the generated samples in the batch, computes its loss on them, and back-propagates accordingly.

However, every 3 iterations, the discriminator samples a batch of images from the cache of previously generated images. This cache includes the batch of images generated from the current iteration, as well as

elements in the cache that were placed there in previous iterations. The discriminator then learns from these randomly sampled generator images.

This wise design choice was done in the original paper and apparently learned from Shrivastava et al, and attempts to alleviate the problem of oscillation in generative adversarial networks. This is a problem where the discriminator only learns to counter the generator's most recent behavior, causing the generator to adopt a previous generating strategy, which results in the discriminator reverting to a previous strategy as well, resulting in unwanted cyclic behavior. Storing a cache of previously generated images alleviates this by ensuring the discriminator performs well on not only the most recent batch of fake images, but previously generated ones.

The way that we did this in version 1 is identical to reference [4]. When the caches are full, the newest images replace a contiguous section of 5 images in the cache. When samples are drawn from the cache, we draw 5 samples which are contiguous in the cache. We upgrade this procedure in versions 2 and 3.

Version 2 Architecture

Gradient penalty

The biggest difference between version 1 and version 2 is the gradient penalty. Our new discriminator objective is the same as the original one with the additional term:

$$\lambda \, \mathbb{E}_{\hat{x} \sim p_{\hat{x}}} [(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]$$

This penalizes large gradients, which should lead to more stable training. We pass an interpolated combination of our fake and real images to the discriminator, and compute the gradient penalty based on the gradient magnitude. The interpolation combines each real image in the batch with its fake one in a uniformly random proportion. We set lambda = 10 following the wgan-gp code reference [9]. (However, "On the Effectiveness of Least Squares Generative Adversarial Networks" [8] suggests using lambda between 30 and 150 for an LSGAN application).

We noticed fluctuating and unstable discriminator losses when we ran this (on the second iteration, loss reached 95). The process of computing loss here involves computing gradients of gradients, and seems to be very memory intensive. The code we used to do this is essentially the same as that in references [9] and [10], and like many other users, we encountered issues with memory. One user solved this by moving the gradient penalty computation to the training code, but this did not help us.

Improved caching

Improved caching during training alters the way that cached fake samples are selected to pass to the discriminator every 3 iterations. In version 1 and reference [4], contiguous samples from the cache are selected to be sent to the discriminator every 3 iterations. However, in version 3 we have made it so that 5 random generated samples are drawn uniformly from the cache of generated images. Drawing 5 contiguous generated samples increases the likelihood of drawing several samples from the same iteration. The cache replacement policy is the same (contiguous) as random sampling obviates the need for changing this.

Different loss ratios
Following a medium article [11], we decide to set lambda to be 10, and weight the cycle loss 10 times more than the adversarial losses, and the identity loss 5 times more. In version 1, this was swapped.

More residually-connected blocks between the generator's encoder and decoder
We increase the number of blocks to be 9, following the tensorflow reference [3].

Version 3 Architecture

This is basically the same as version 2, but we removed the gradient penalty due to memory limits and issues.

Results and discussion

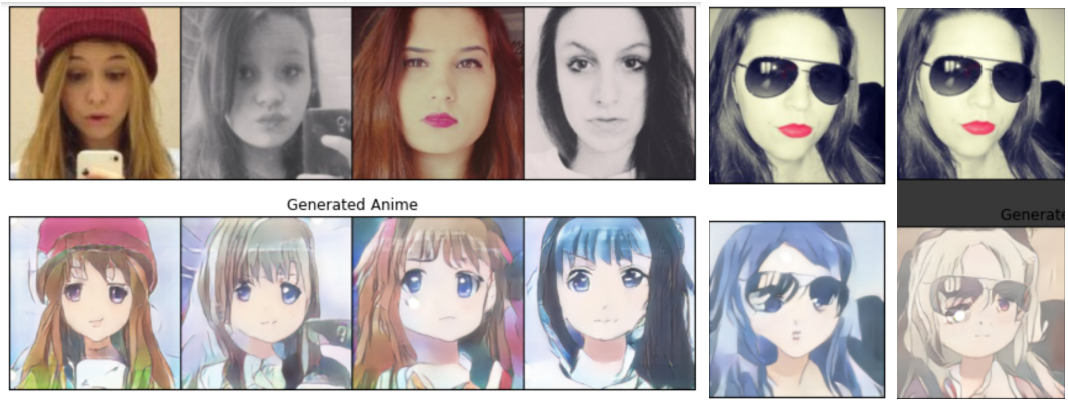
We trained the baseline model for 86 complete epochs, which took 24 hours. After 24 hours, we got disconnected from the runtime. We trained version 1 for 56 epochs, and version 3 for 75 epochs. Version 2 runs out of memory within 12 iterations (88% of the first epoch).

Our generator occasionally succumbs to mode collapse. The cycle-consistency loss does not appear to fully eliminate mode collapse, despite imposing the requirement of image reconstruction. In version 3 we doubled the loss weight on the cycle loss, and also sometimes saw mode collapse. Training was slow (about 15 minutes per epoch for all models), which made searching the hyperparameter space beyond obvious or suggested improvements difficult. Also, the task of generating realistic faces from anime images appears to be more difficult than generating anime faces from real ones. Our model was much slower at learning this, and did not quite get there.

We have included 50 pairs of selfies and their translations to anime faces at the end of this report. They are also visualized as a grid by the "Model Loader and Tester v1" and "Model Loader and Tester v3" ipynb notebooks, as suggested by the project requirement write-up (downloading models from our Drive will be necessary for new visualizations.

Below are some specific observations along with cool examples:

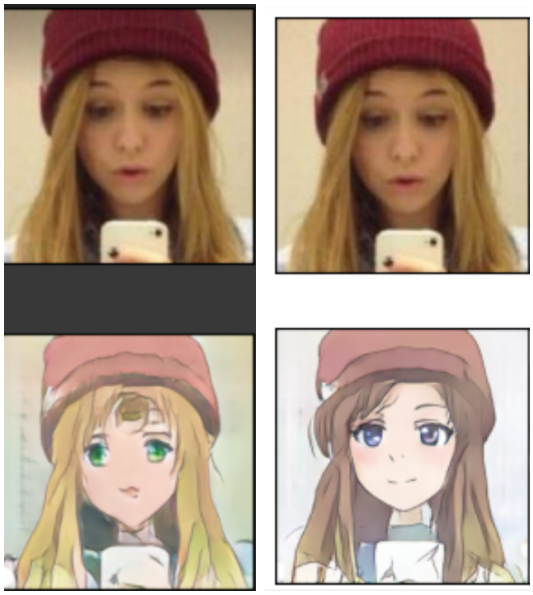
Generated images sometimes preserve objects in the scene



In the first two photos in this example, the phones are preserved (and the hat in the first one). Additionally, we observe that black-and-white images are colorized. Images generated from the test set during training, using version 3, except for the last example which is from version 1.

Different Models colorize images differently

Here is an example where the first image above was colored differently (below), with green eyes and lighter hair. Images from version 1 and version 3 generated from the test set during training.



More examples are below the references. We display 50 images for versions 3 and 1 of our model.

References

Link to our Drive folder:
https://drive.google.com/drive/folders/1cEyxQwM9FpVmMR7np_a_7xuNDE9u5TI-?usp=sharing

[1] Original Cycle-GAN paper, "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks": https://arxiv.org/pdf/1703.10593.pdf

[2] Baseline Model: https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix

[3] Reference for Discriminator Architecture:
https://machinelearningmastery.com/cyclegan-tutorial-with-keras/

[4] Reference for Generator Architecture and Training Strategy.
Used this code to help test/plot, load data, and save and load models:
https://towardsdatascience.com/cycle-gan-with-pytorch-ebe5db947a99

[5] BatchNorm vs Instance Norm
https://medium.com/techspace-usict/normalization-techniques-in-deep-neural-networks-9121bf100d8

[6] Reference for different types of GAN Loss:
https://jonathan-hui.medium.com/gan-does-lsgan-wgan-wgan-gp-or-began-matter-e19337773233

[7] Gradient Penalty (Improved Training of Wasserstein GANs)
https://arxiv.org/pdf/1704.00028.pdf

[8] LSGANs with gradient penalty, "On the Effectiveness of Least Squares Generative Adversarial Networks":
https://arxiv.org/pdf/1712.06391.pdf

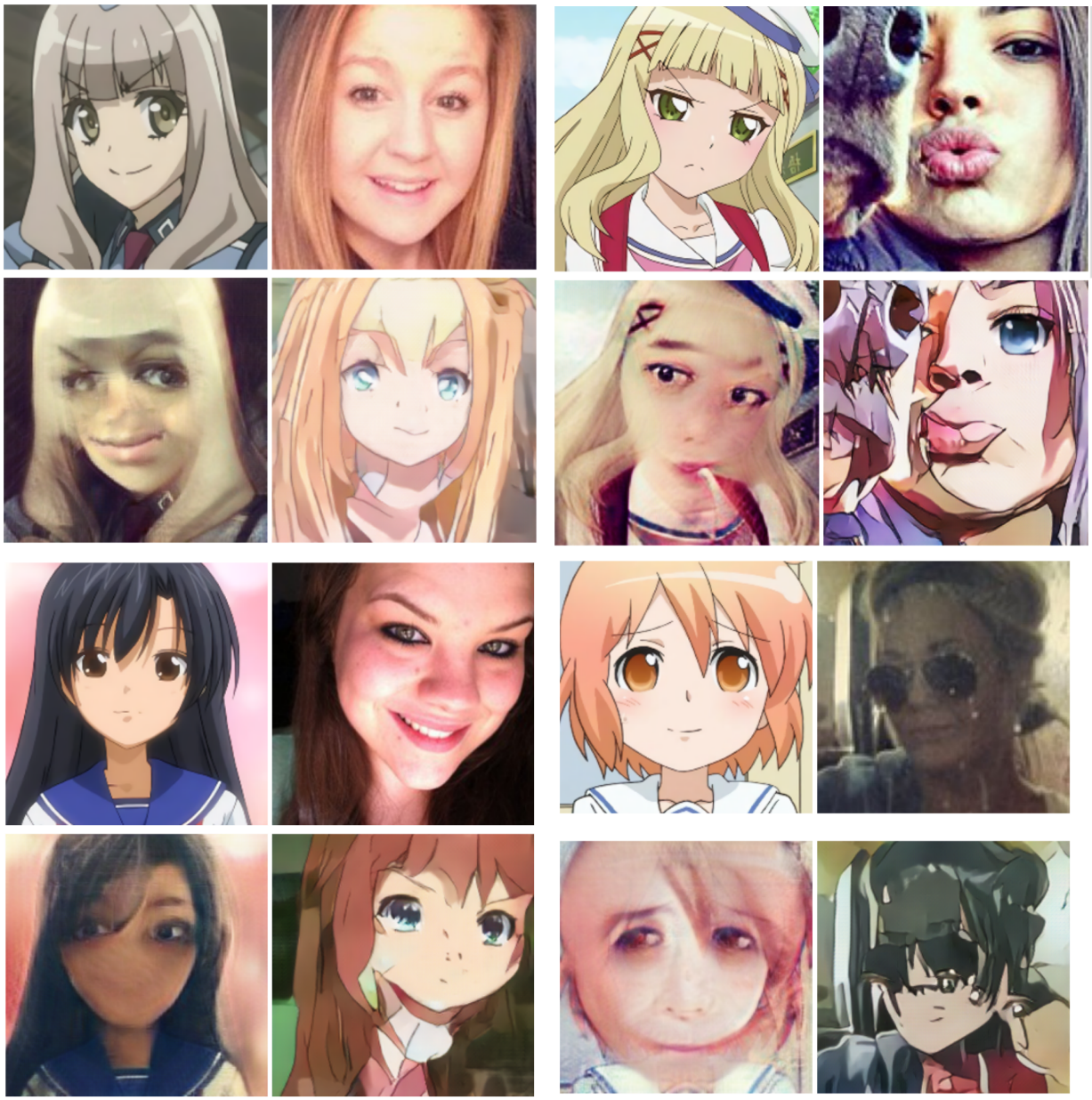
[9] Reference code 1 for computing gradient penalty:
https://github.com/caogang/wgan-gp/blob/master/gan_mnist.py

[10] Reference code 2 for computing gradient penalty:
https://towardsdatascience.com/demystified-wasserstein-gan-with-gradient-penalty-ba5e9b905ead

[11] Idea for changing the loss weights on the different components of generator loss:
https://medium.com/analytics-vidhya/the-beauty-of-cyclegan-c51c153493b8

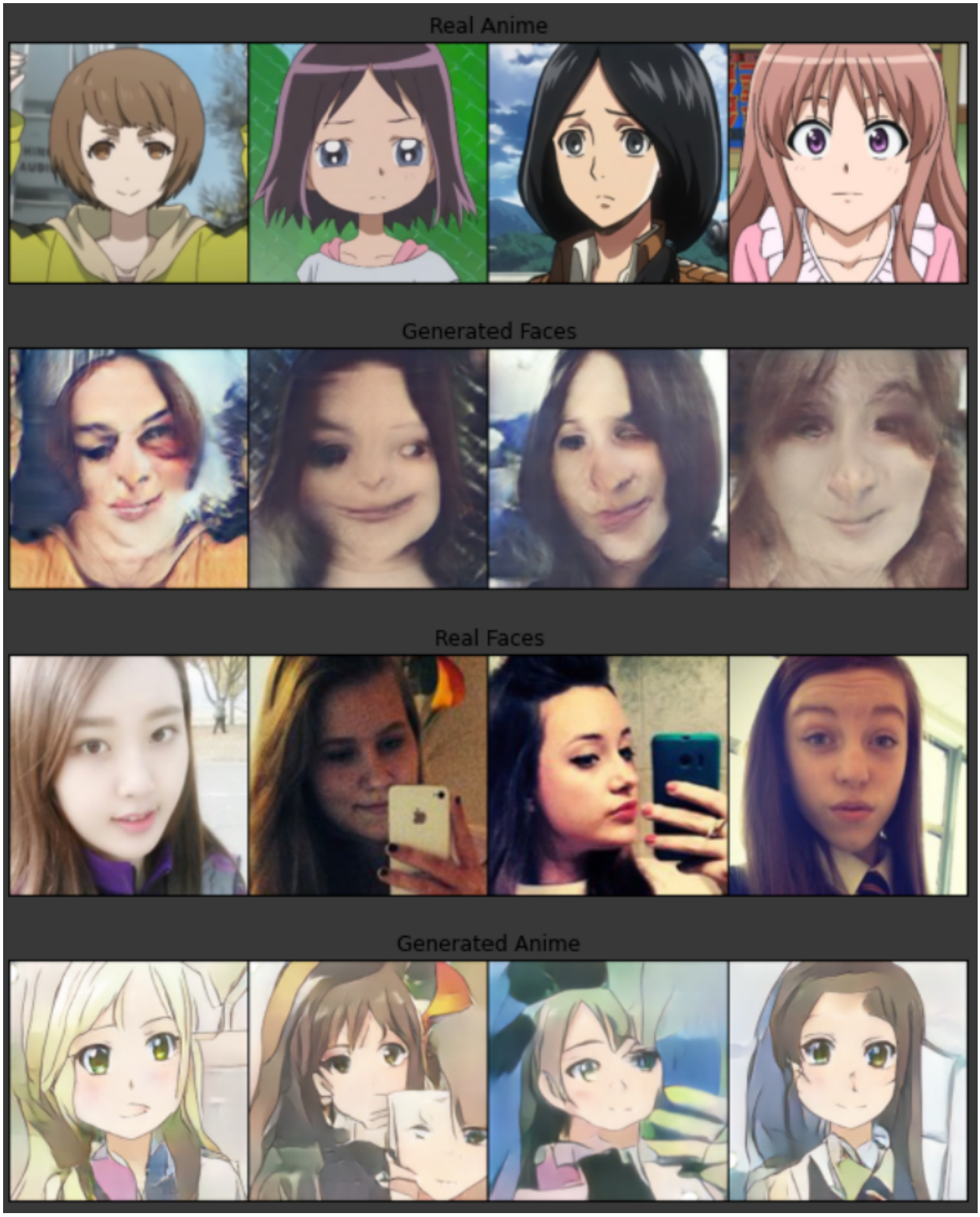
Example Images

Baseline (86 epochs). Images are paired with their translation below.



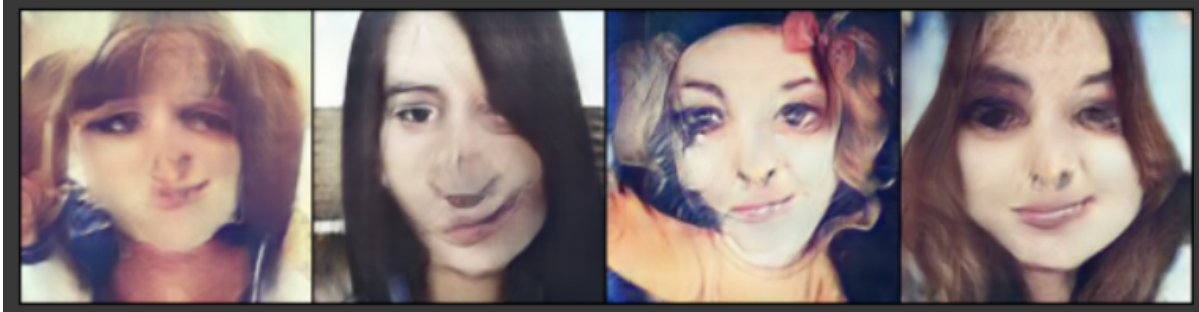
Version 3 (68 Epochs)

We trained version 3 for 75 Epochs, but believe we achieved our best results on Epoch 68. (See next page)

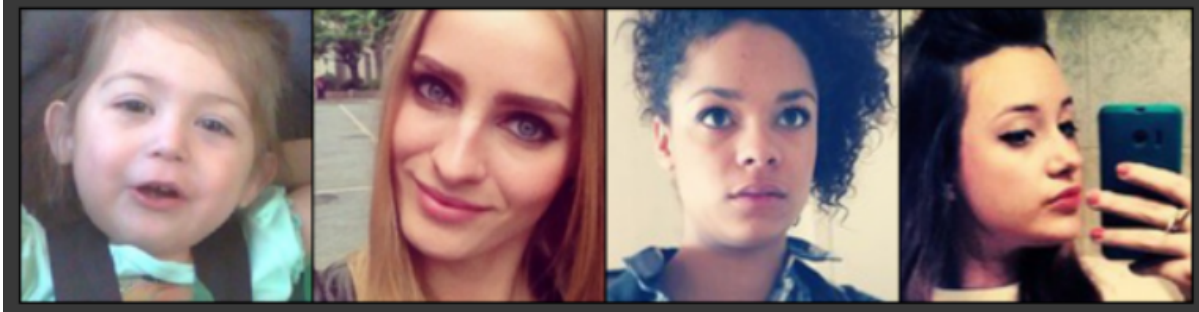

Real Anime
Generated Faces
Real Faces
Generated Anime

Real Anime

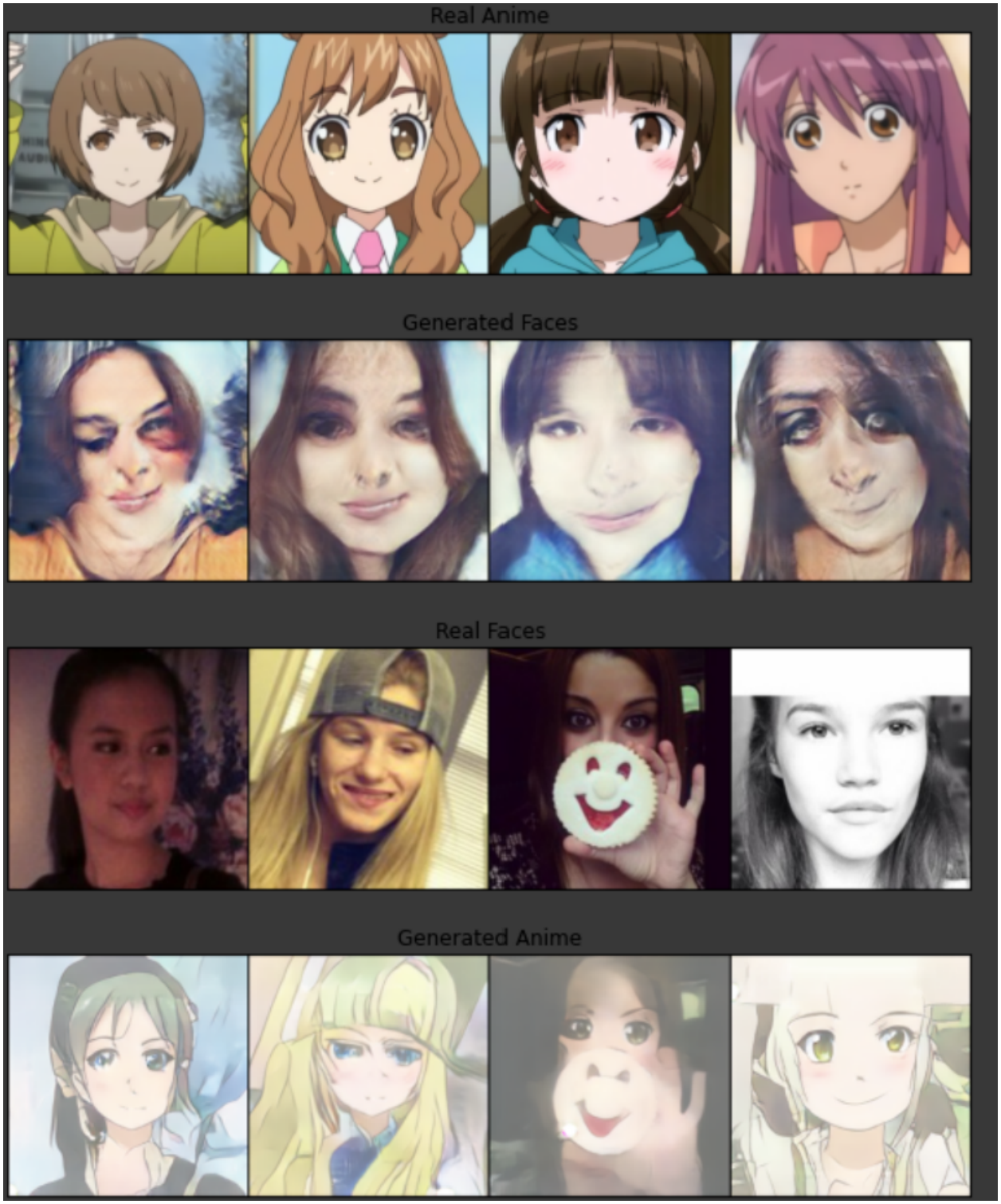
Generated Faces



Real Faces

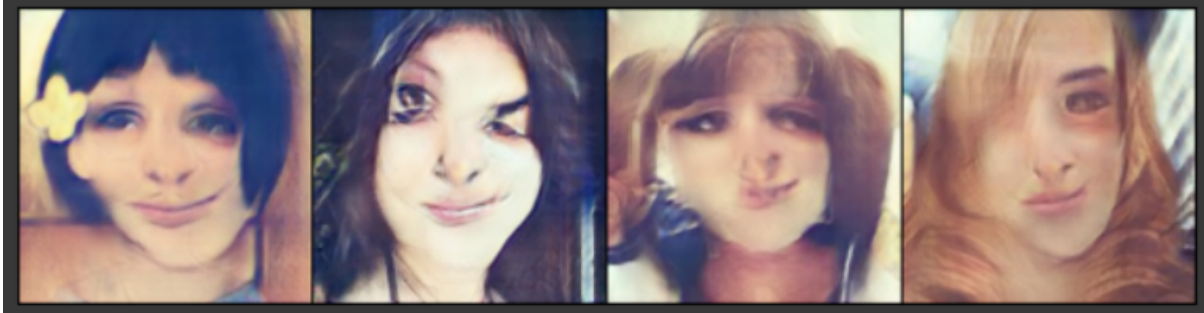


Generated Anime


Real Anime
Generated Faces
Real Faces
Generated Anime

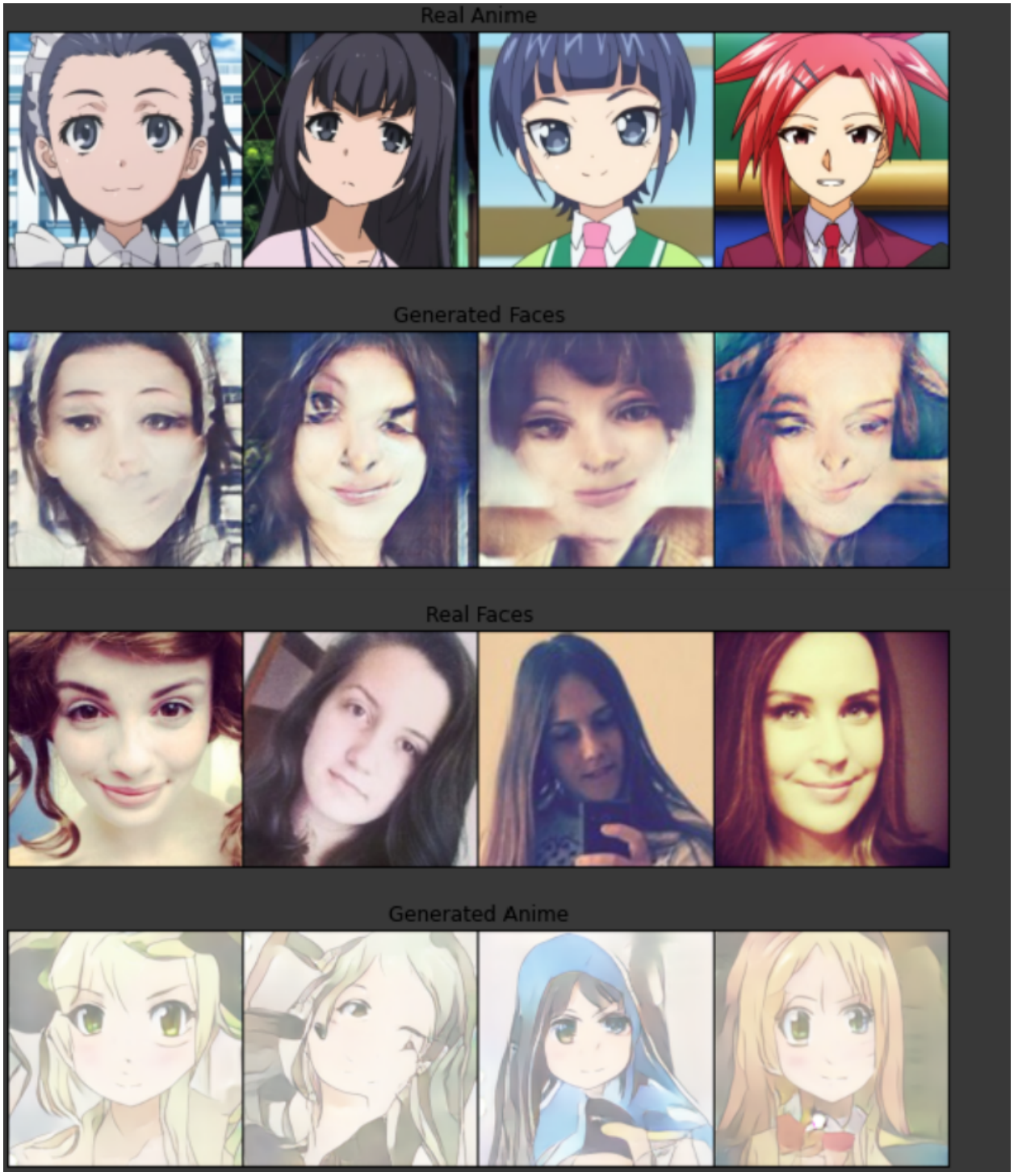
Real Anime

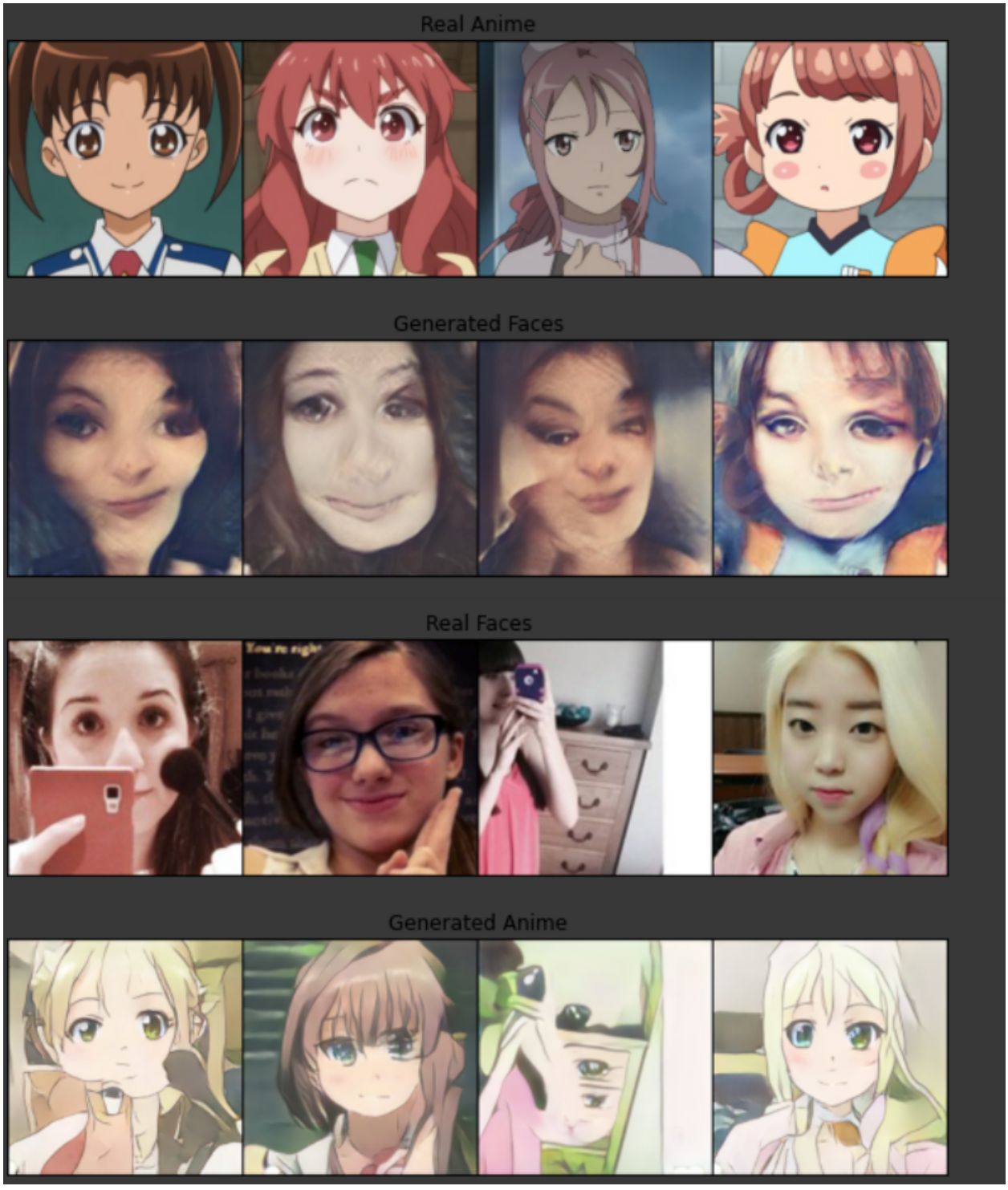
Generated Faces

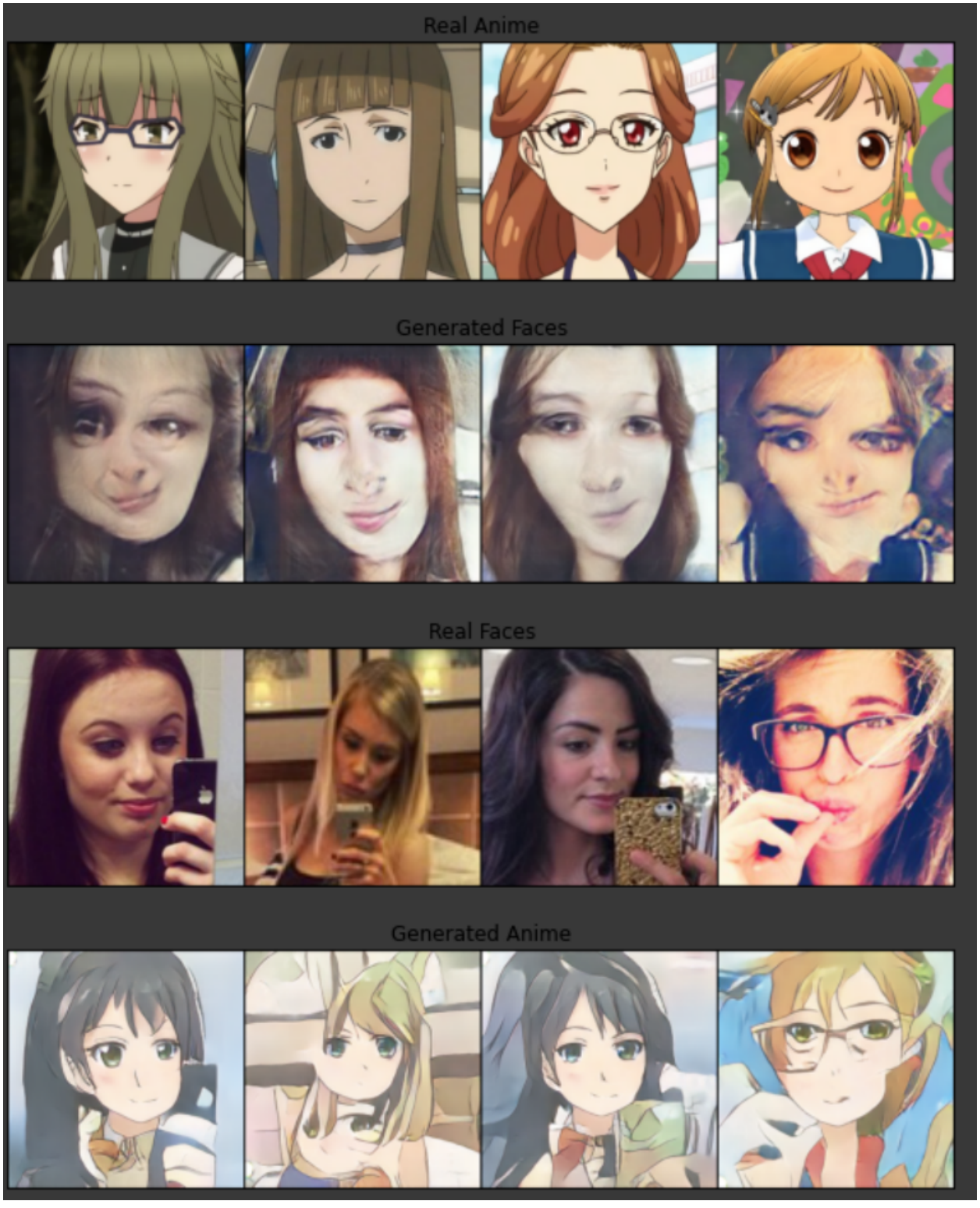


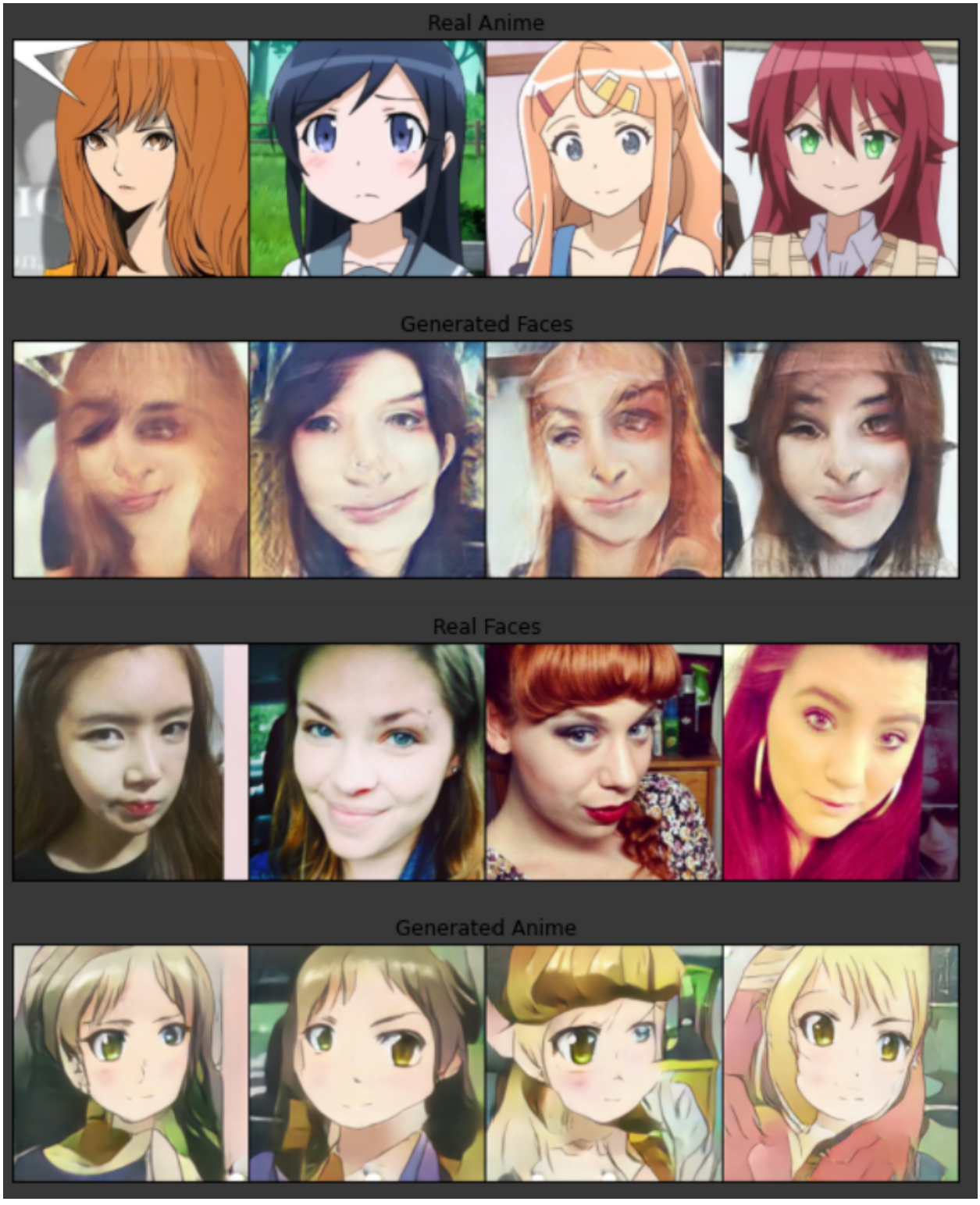
Real Faces

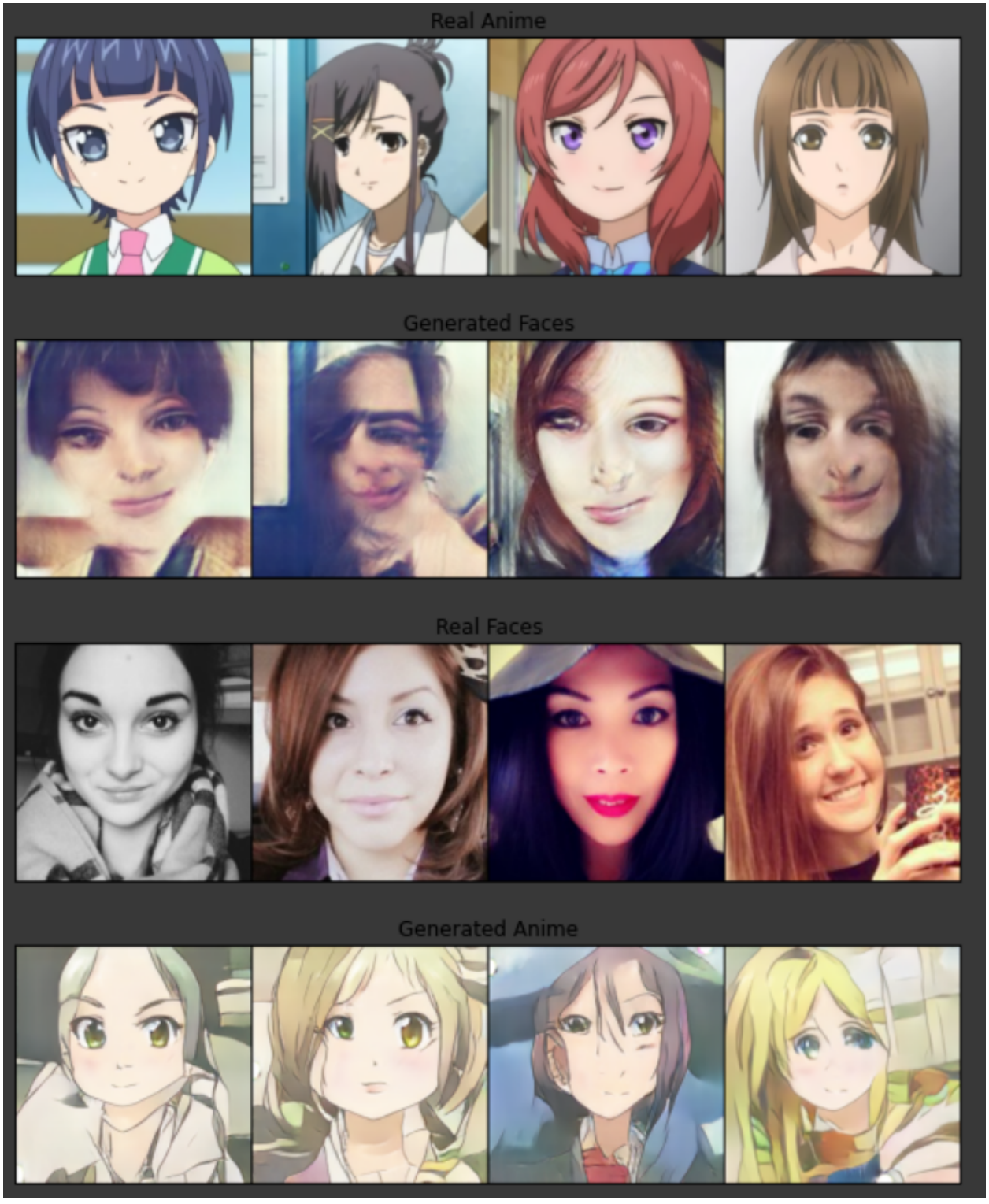
Generated Anime

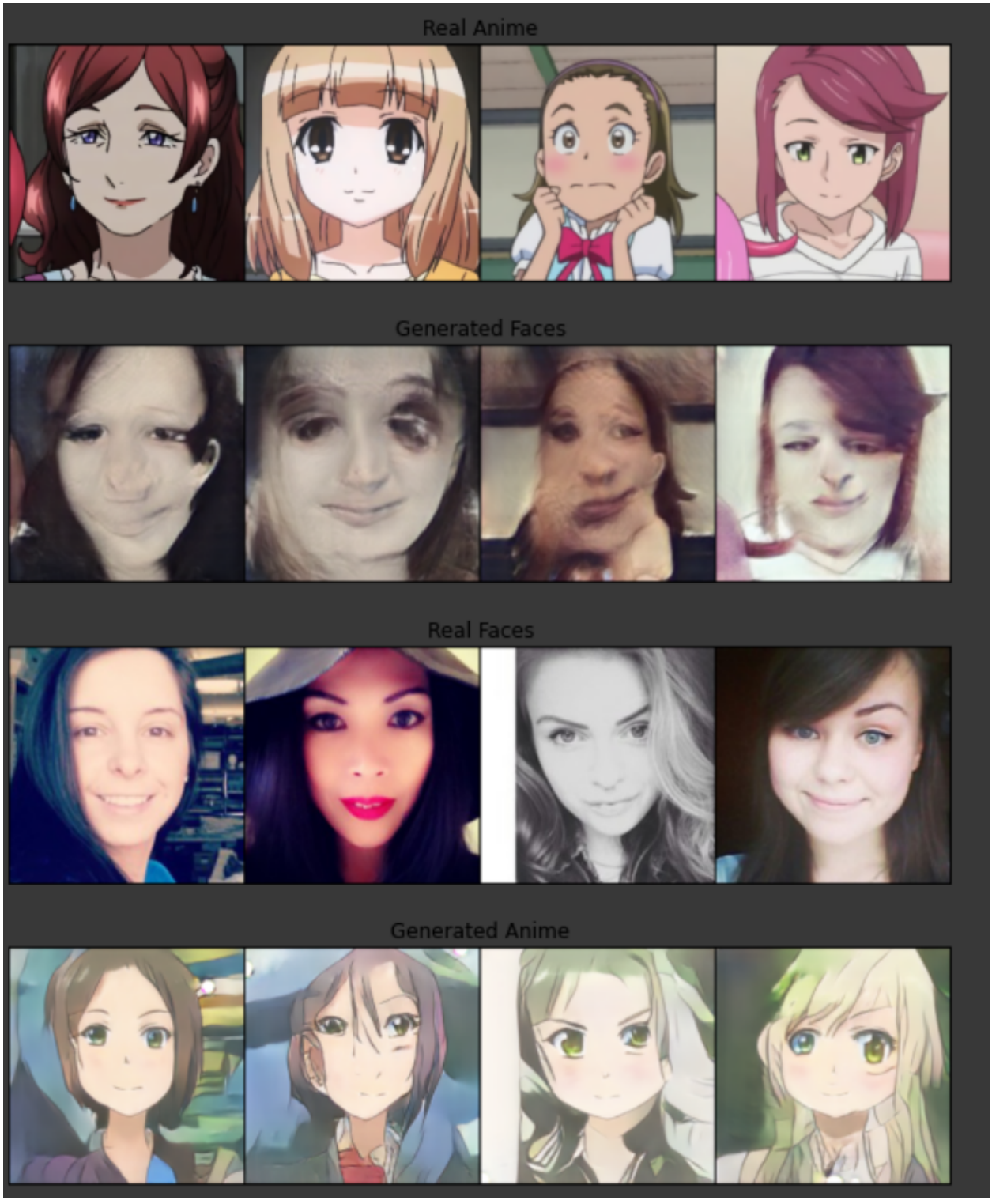

Real Anime
Generated Faces
Real Faces
Generated Anime


Real Anime
Generated Faces
Real Faces
Generated Anime


Real Anime
Generated Faces
Real Faces
Generated Anime


Real Anime
Generated Faces
Real Faces
Generated Anime

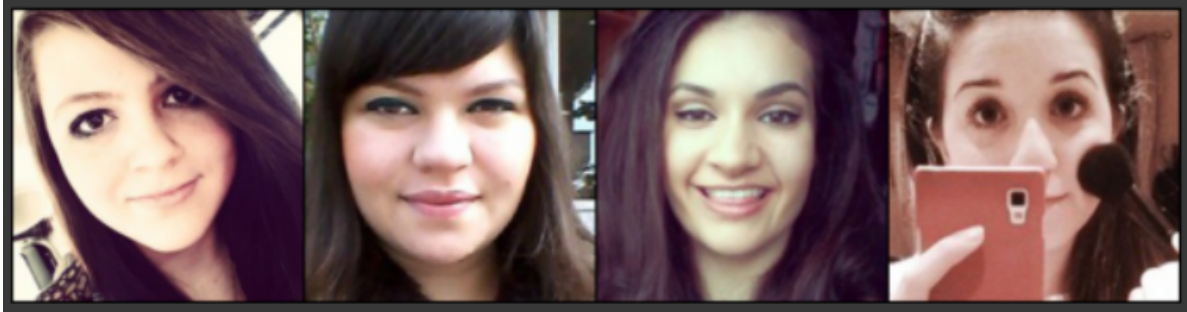

Real Anime
Generated Faces
Real Faces
Generated Anime


Real Anime
Generated Faces
Real Faces
Generated Anime

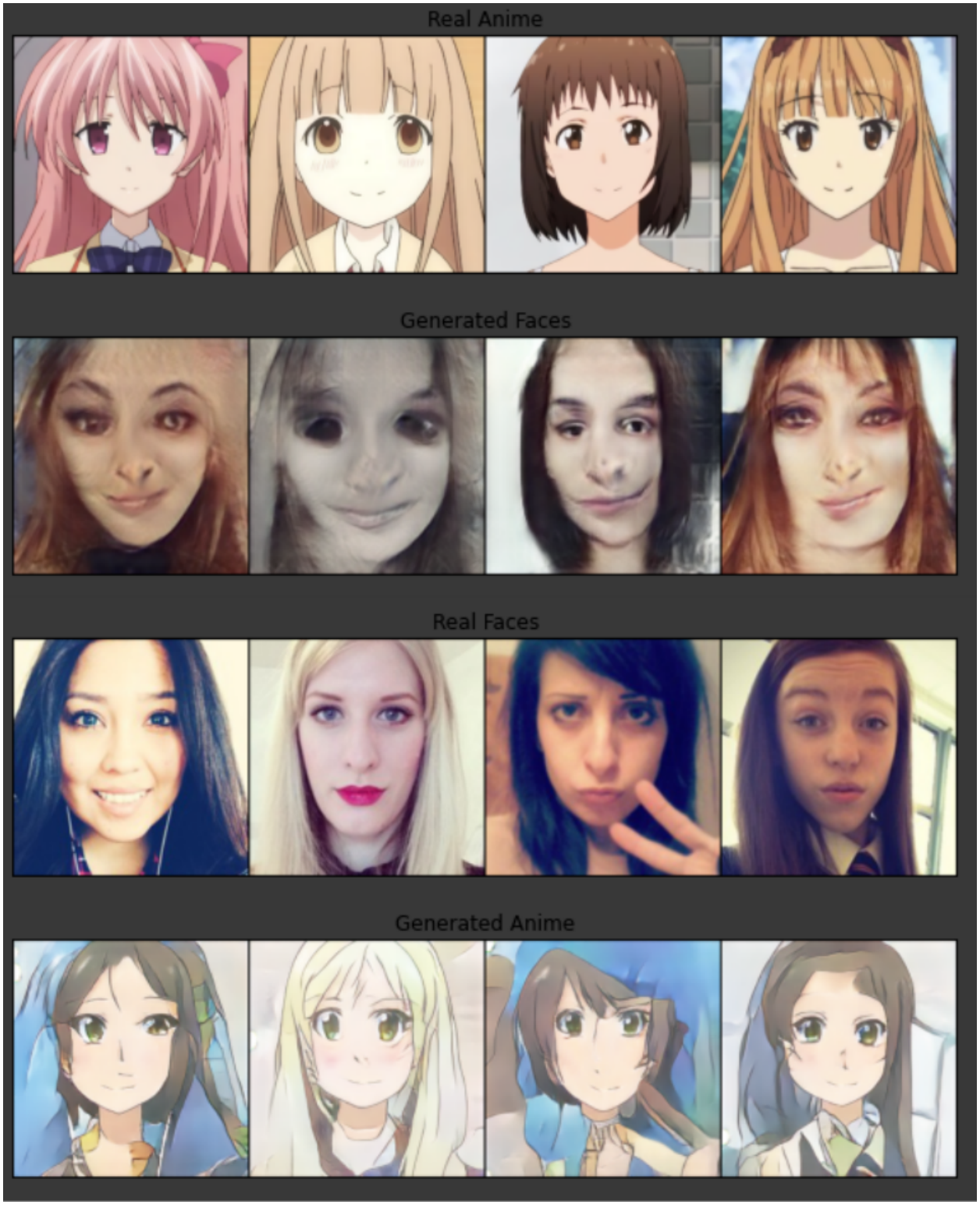
Real Anime

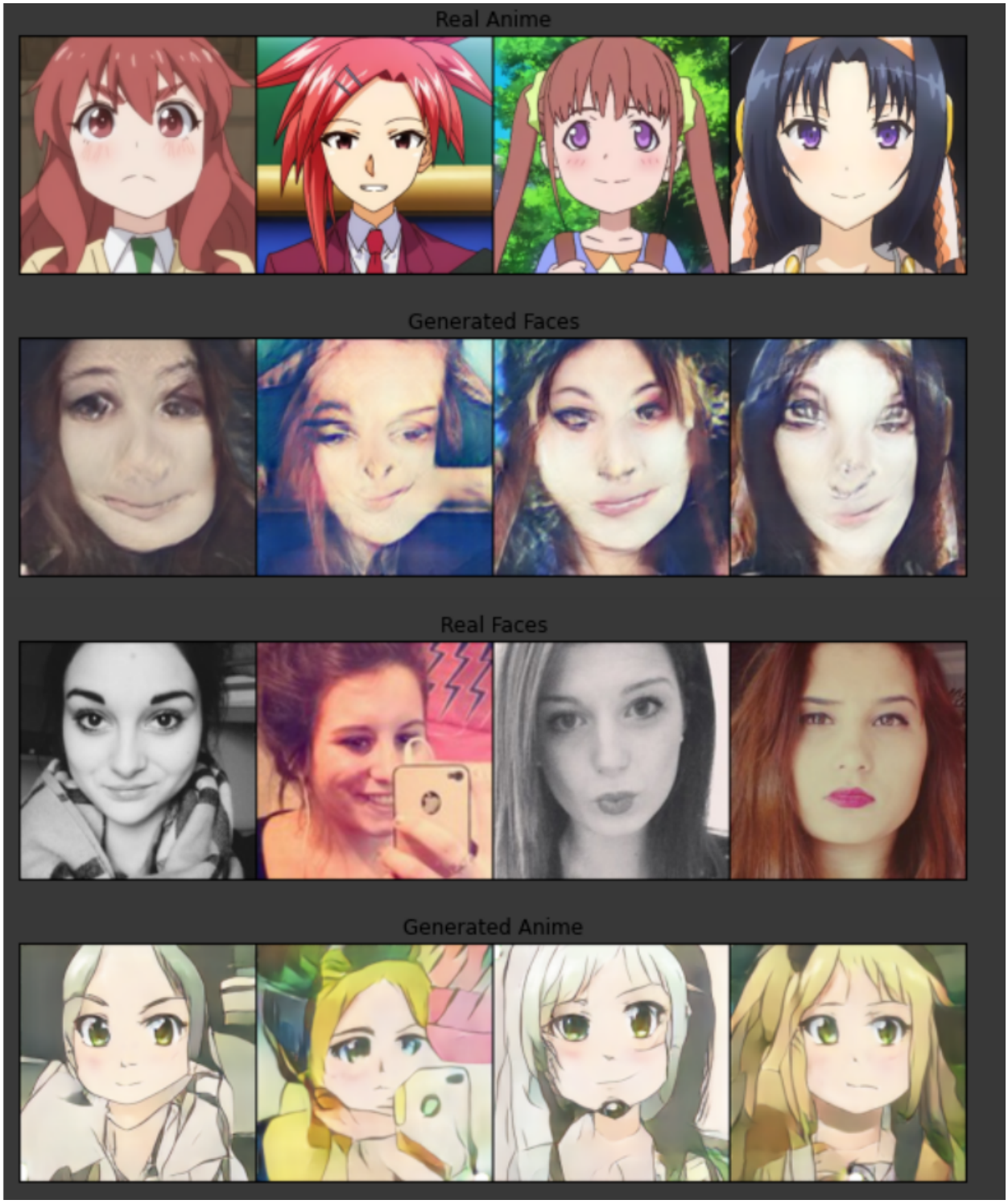
Generated Faces

Real Faces



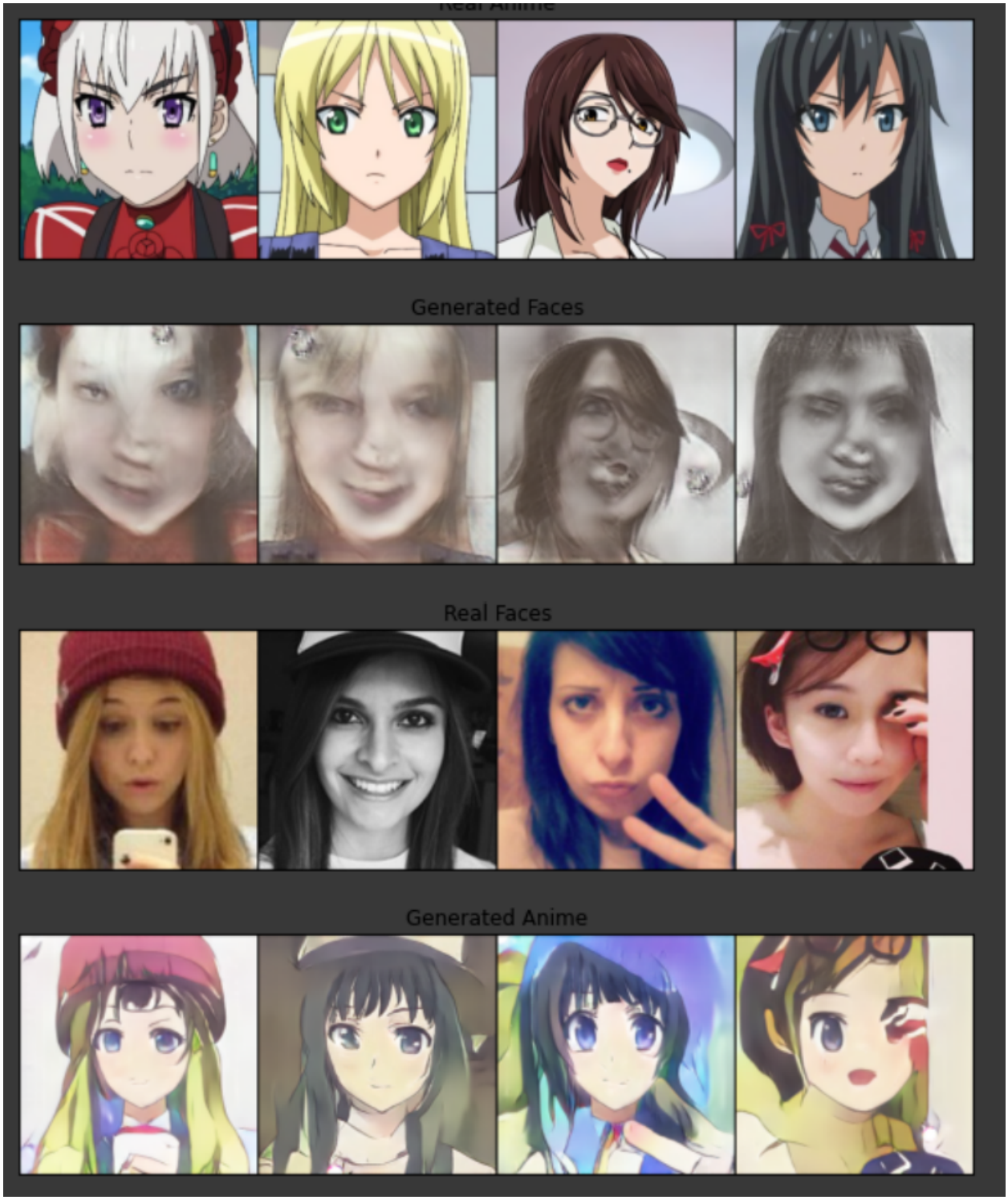
Generated Anime

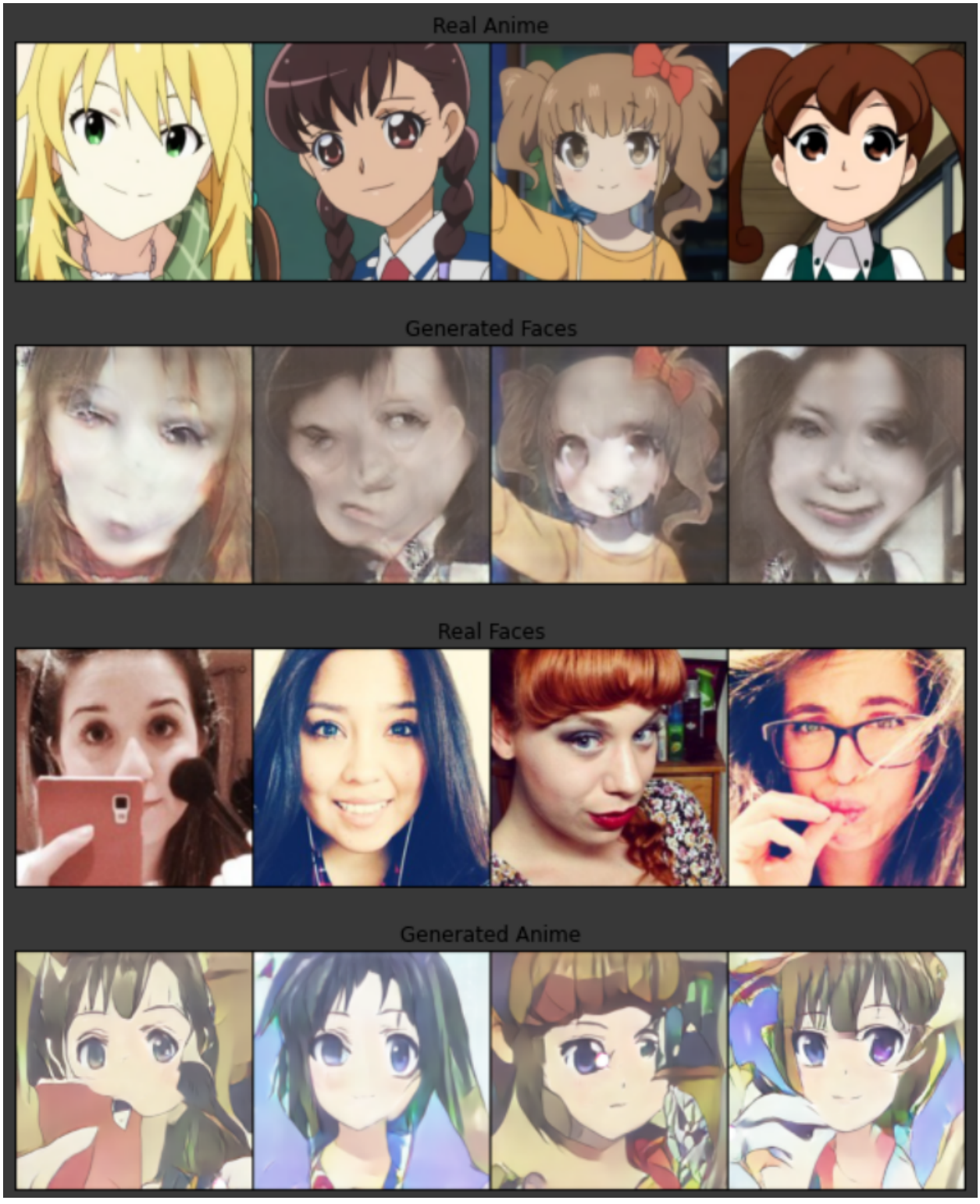

Real Anime
Generated Faces
Real Faces
Generated Anime

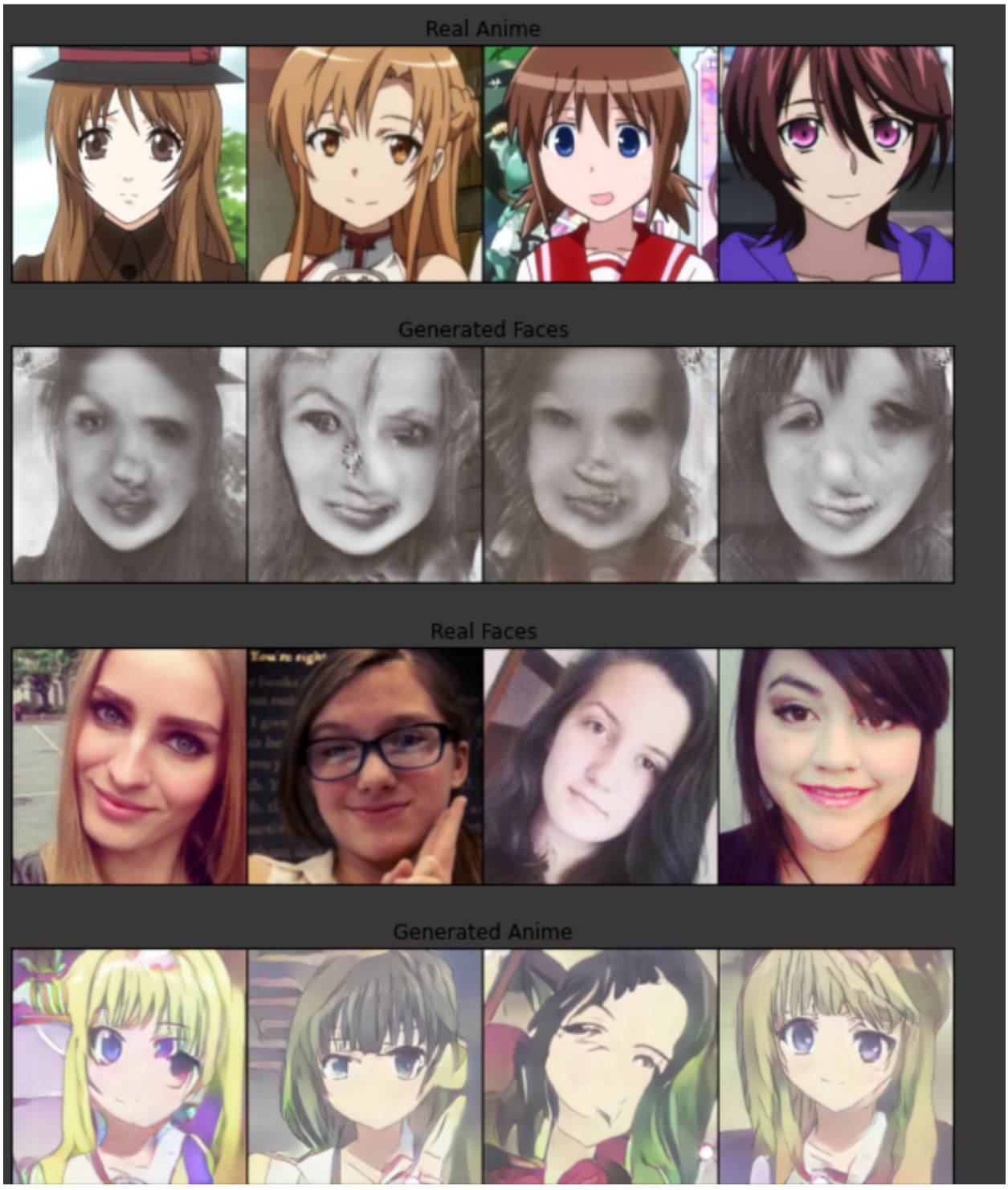

Real Anime
Generated Faces
Real Faces
Generated Anime

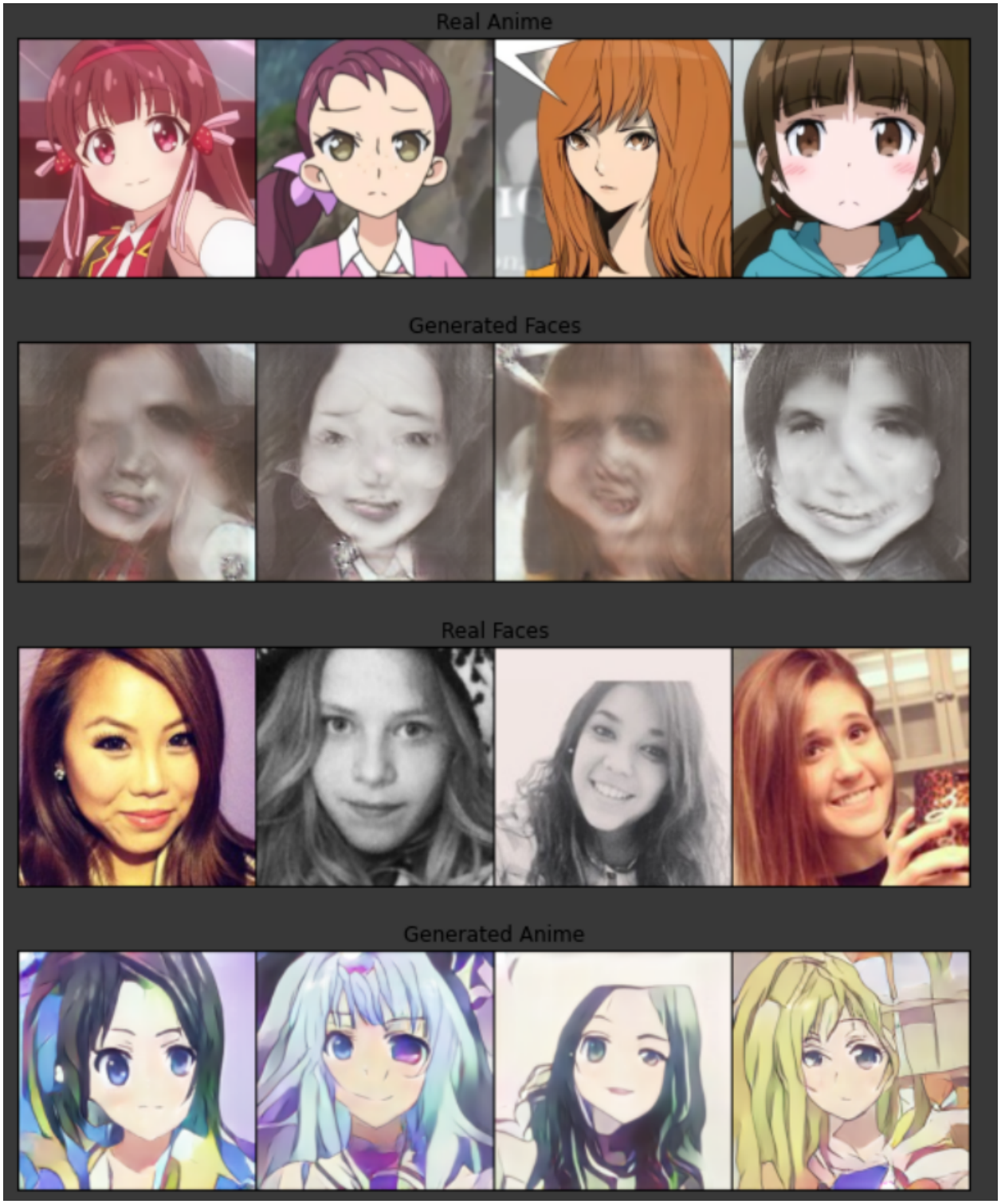
Version 1 (After 56 Epochs)

The best images seemed to be generated at the end of 56 epochs, which is also how long we trained the model for in total.

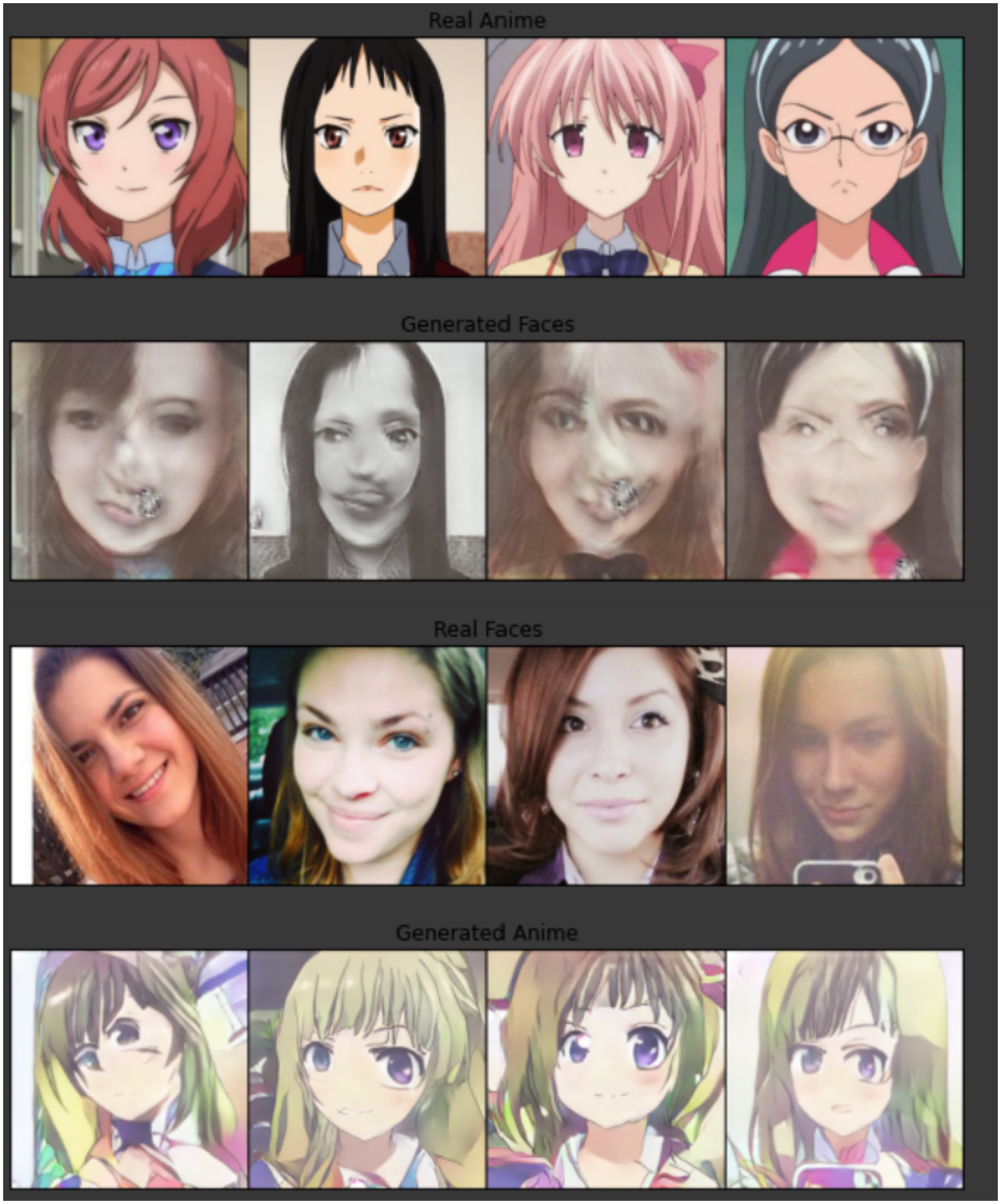


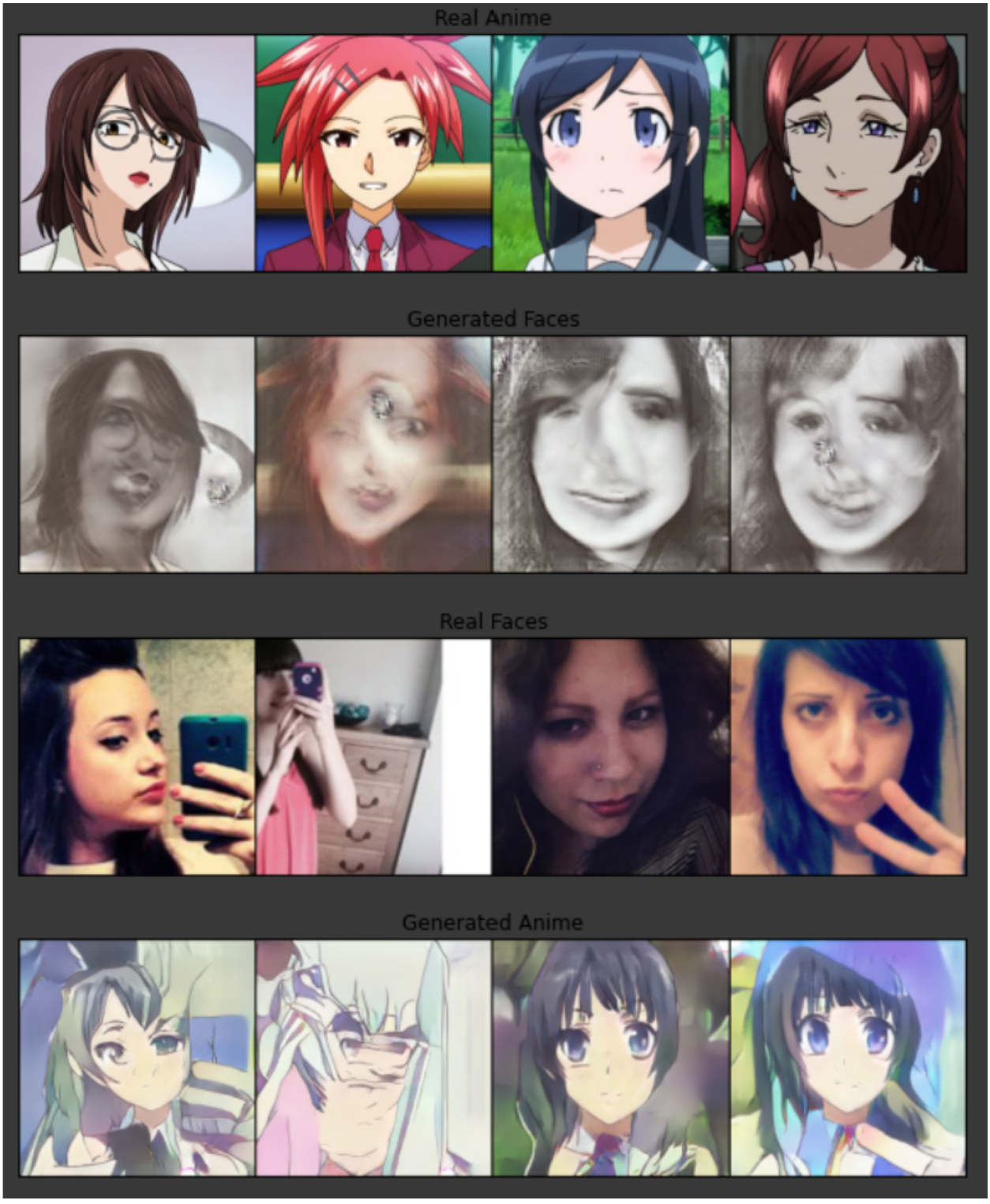

Real Anime
Generated Faces
Real Faces
Generated Anime

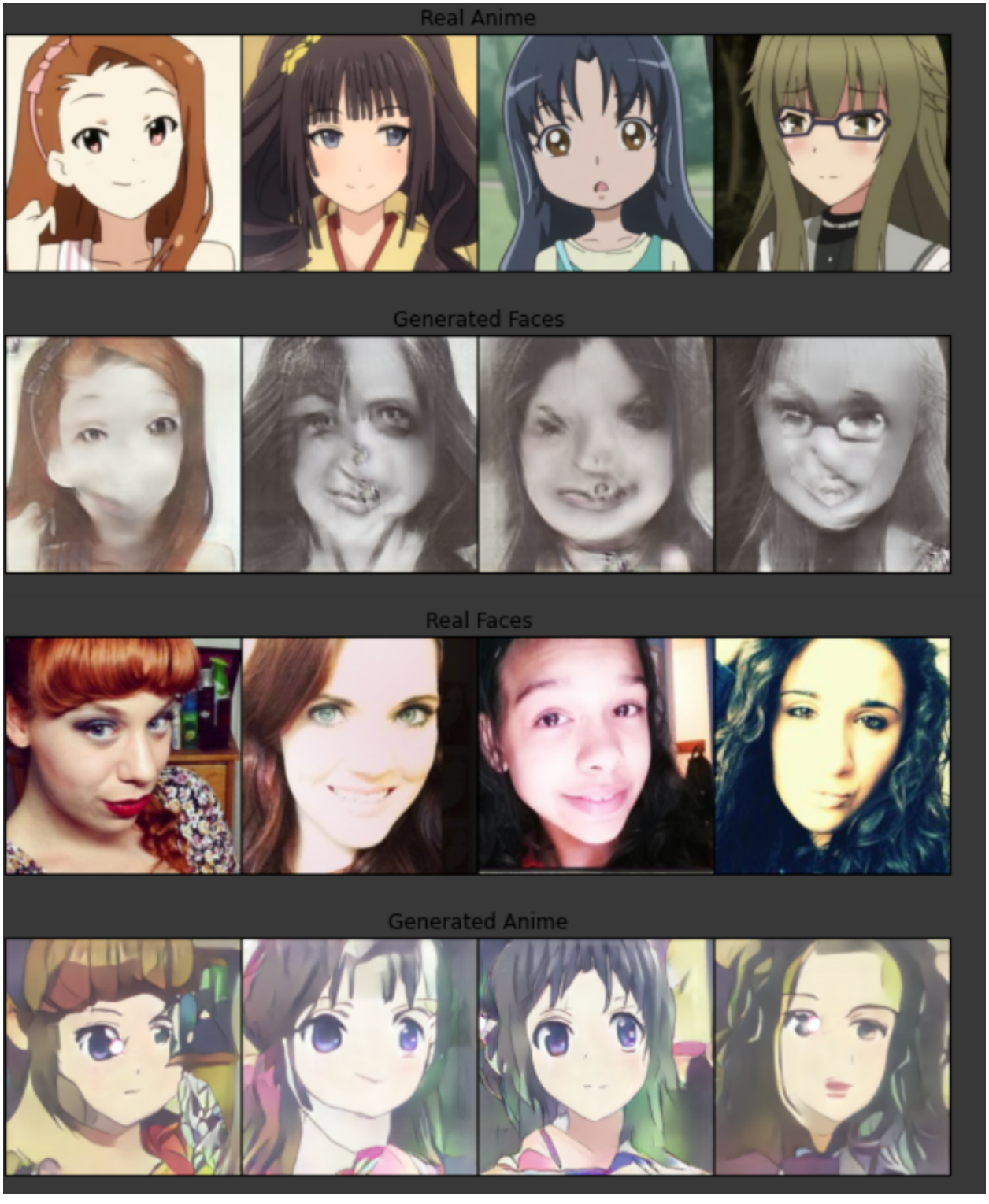

Real Anime
Generated Faces
Real Faces
Generated Anime

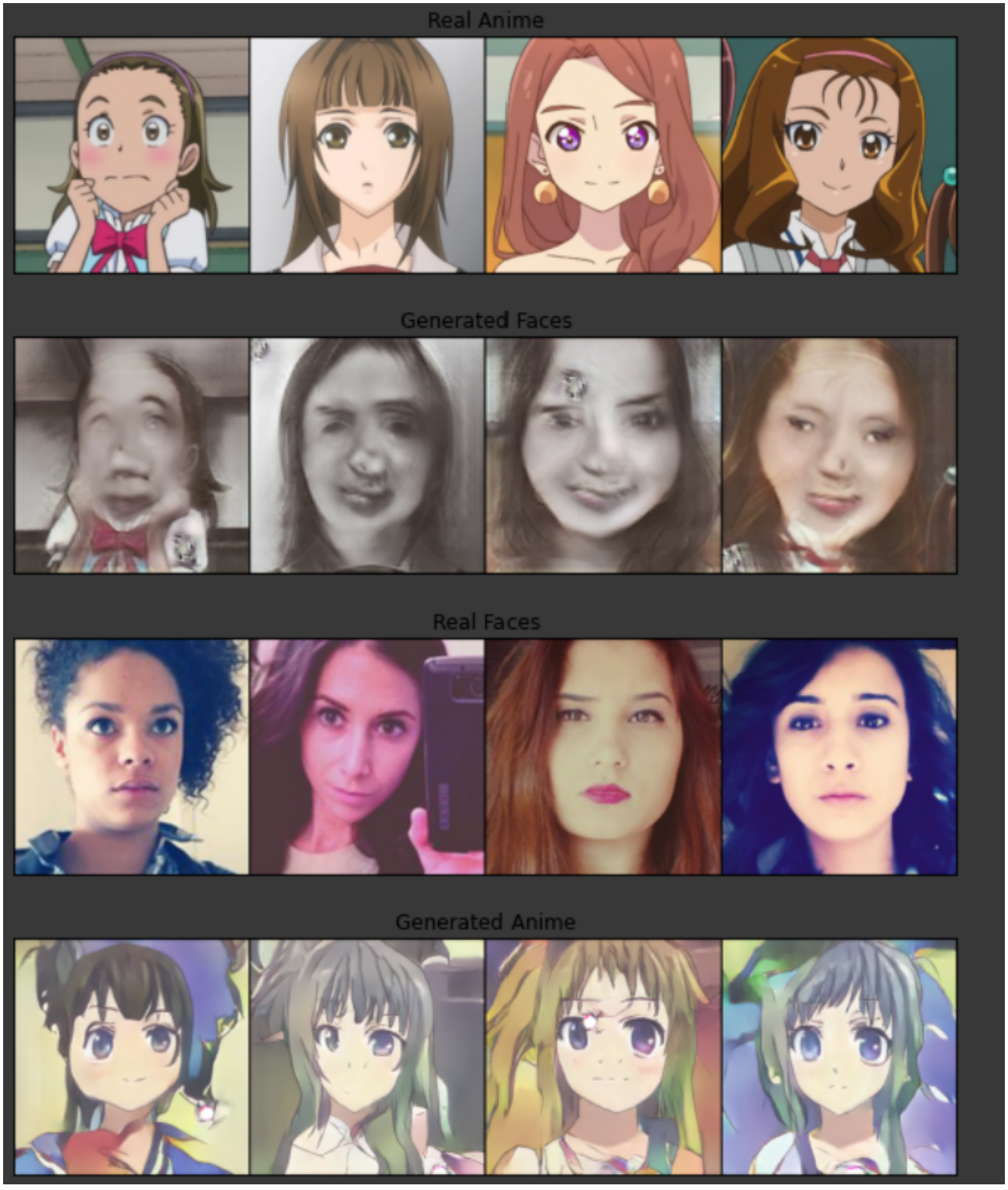

Real Anime
Generated Faces
Real Faces
Generated Anime

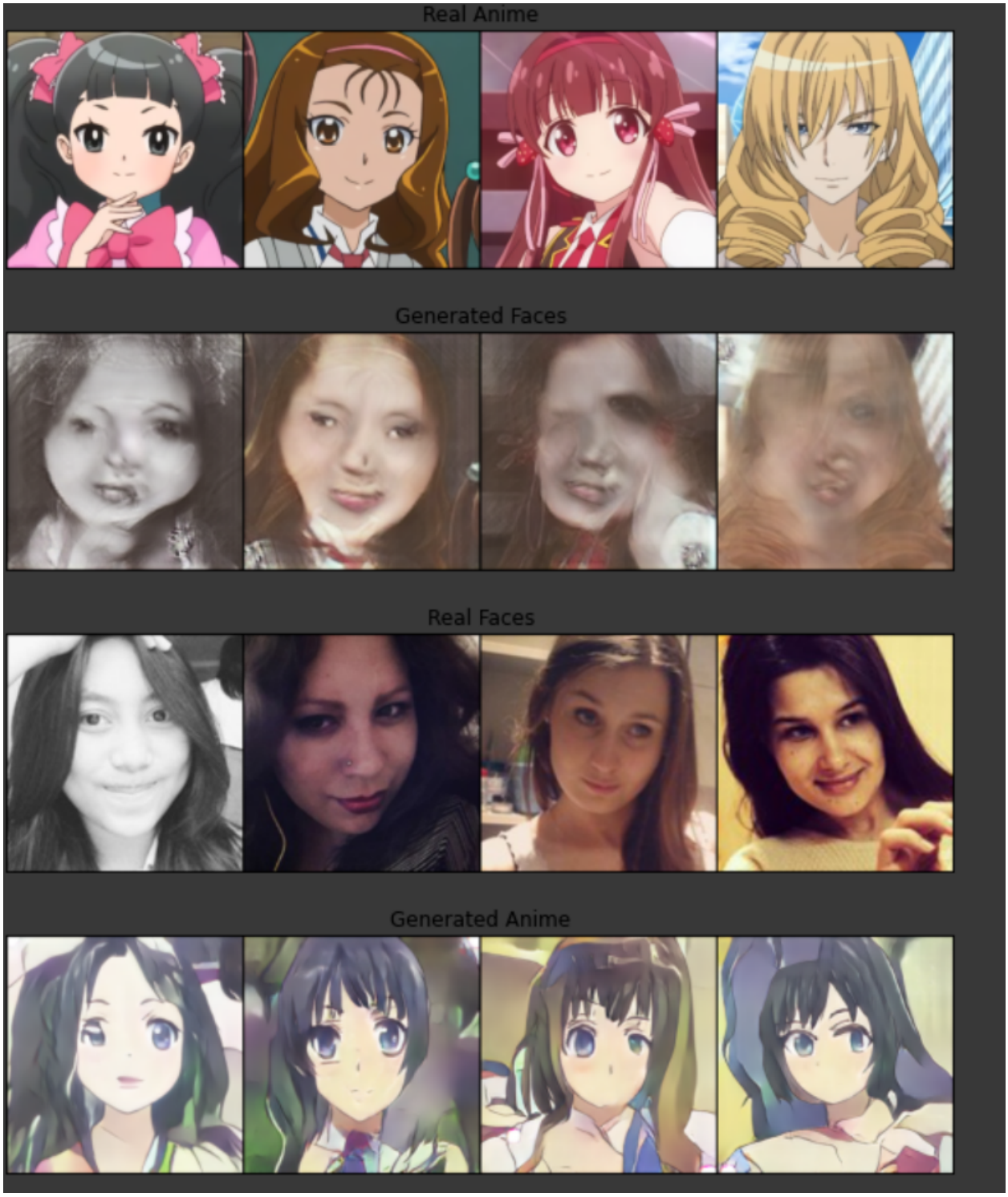

Real Anime
Generated Faces
Real Faces
Generated Anime

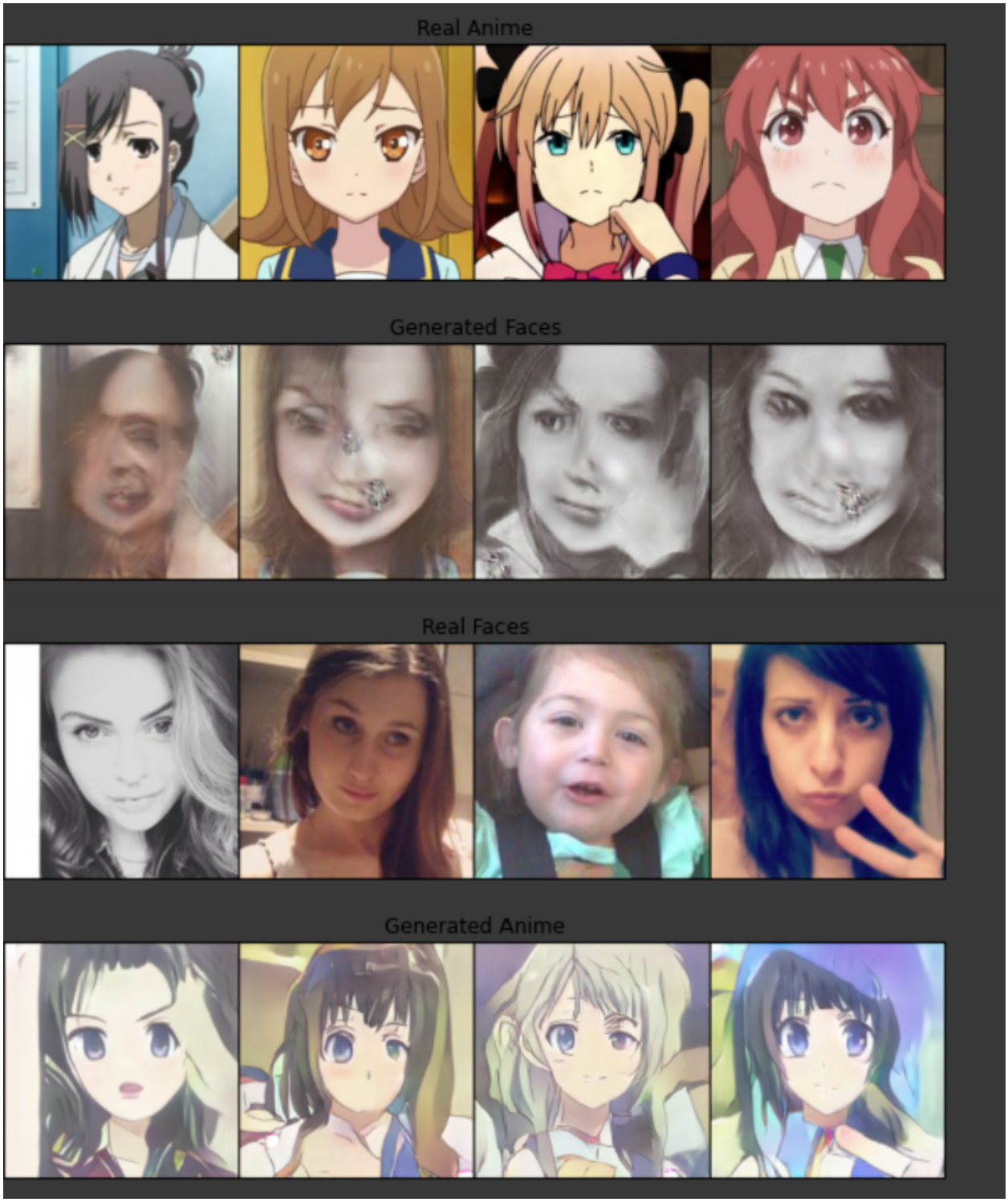

Real Anime
Generated Faces
Real Faces
Generated Anime

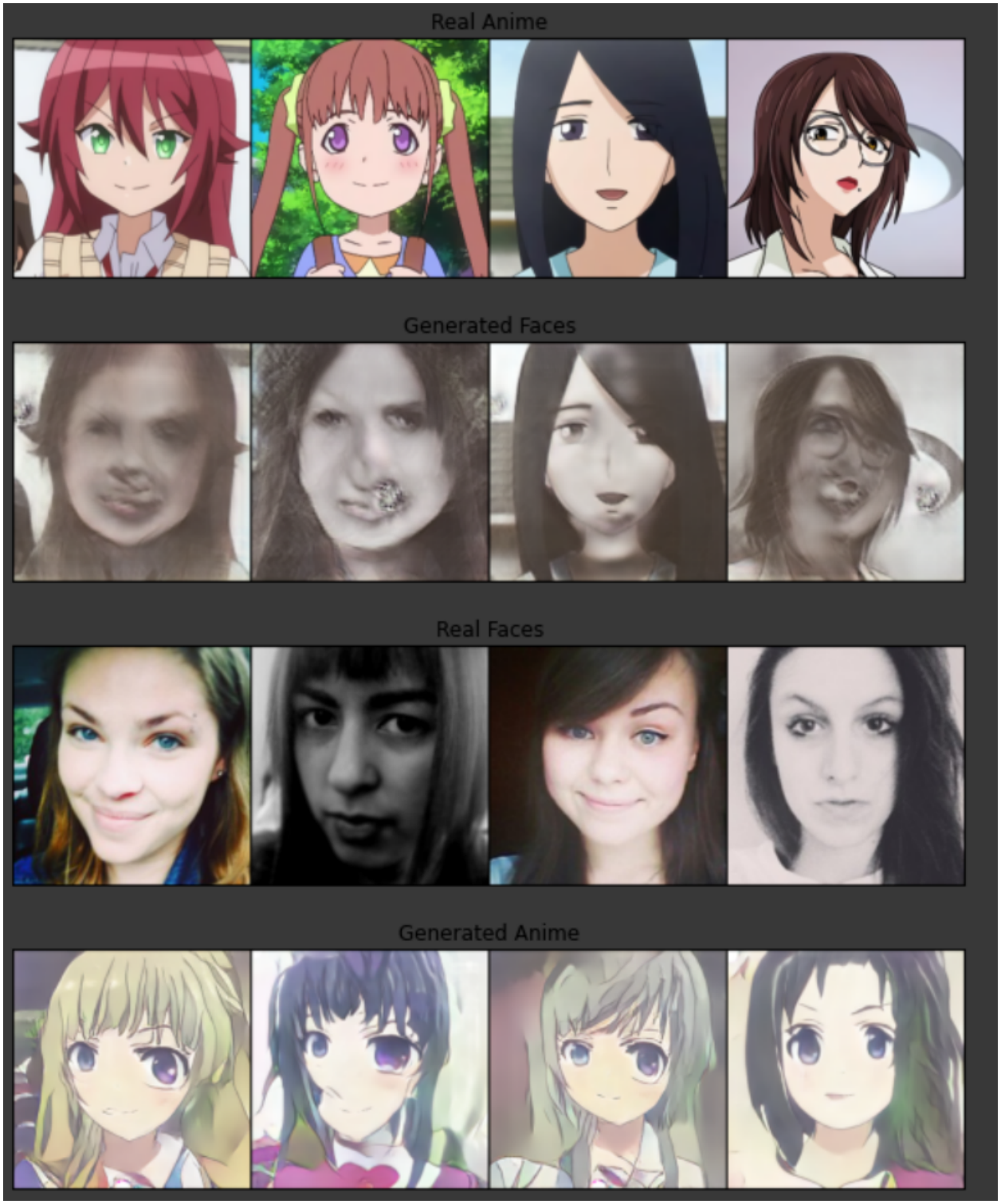

Real Anime
Generated Faces
Real Faces
Generated Anime

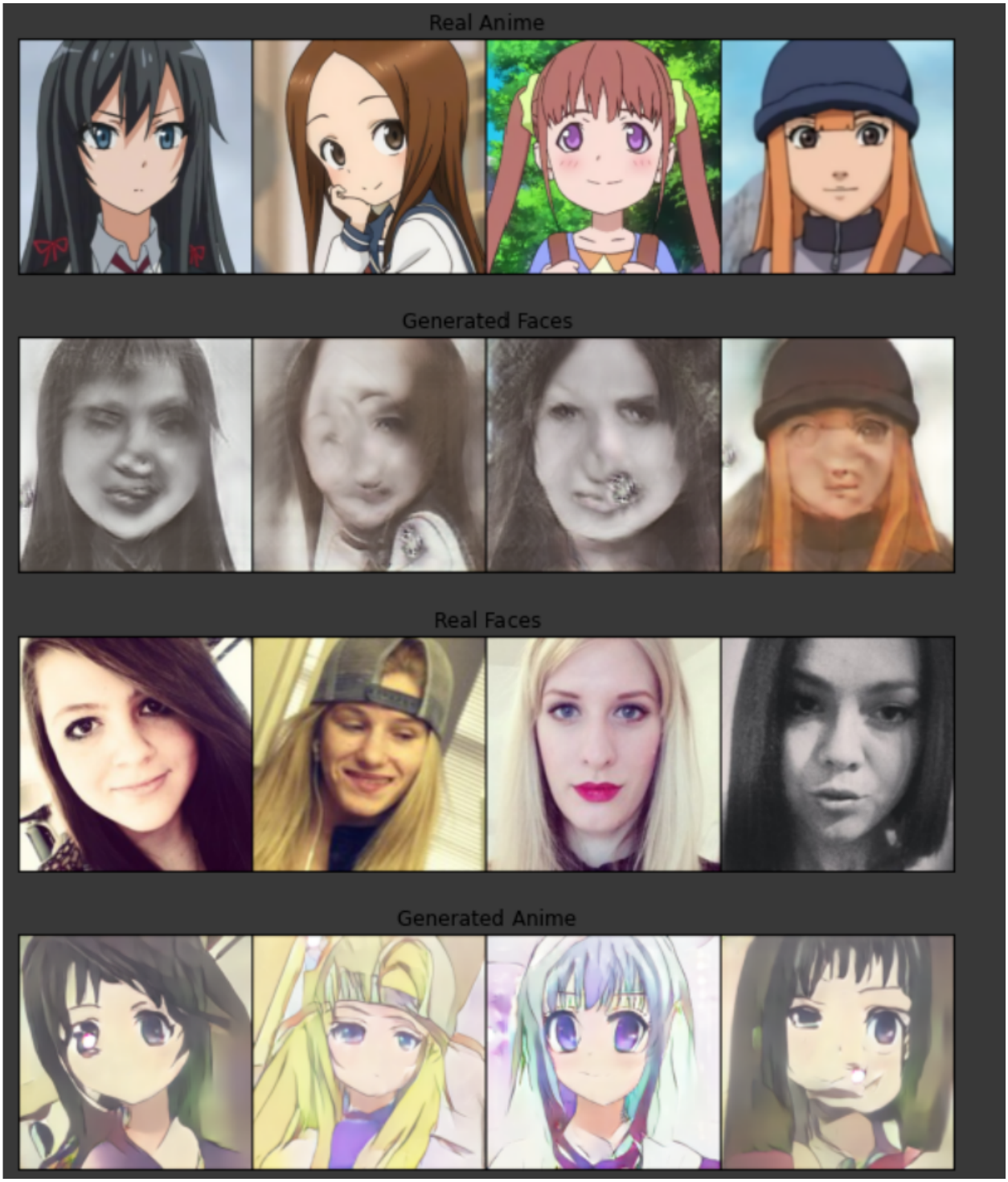

Real Anime
Generated Faces
Real Faces
Generated Anime

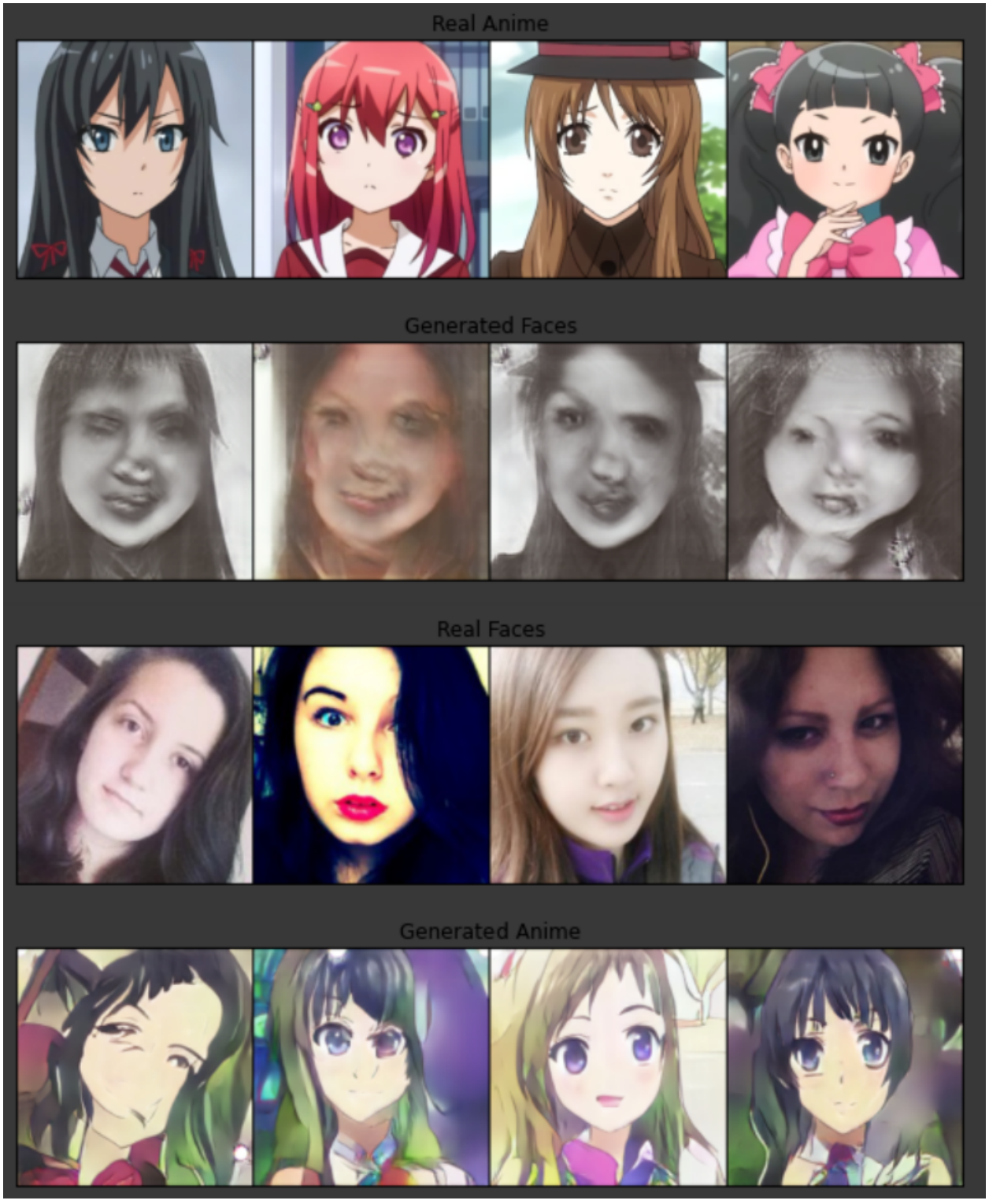

Real Anime
Generated Faces
Real Faces
Generated Anime


Real Anime
Generated Faces
Real Faces
Generated Anime


Real Anime
Generated Faces
Real Faces
Generated Anime


Real Anime
Generated Faces
Real Faces
Generated Anime


Real Anime
Generated Faces
Real Faces
Generated Anime


Real Anime
Generated Faces
Real Faces
Generated Anime