# GAN On Anime Faces Dataset

Thi Thu Nhi, Nguyen
Peppi ID: 132187
Thinguy21@student.oulu.fi

## Abstract

*Following the publication of the Generative Adversarial Network (GAN), researchers focused on the automatic generation of facial images. There have been some attempts to apply the GAN model to the problem of generating facial images of anime characters, but none have yielded promising results. In this report, I investigate the training of GAN models on an anime facial image dataset.*

## 1. Introduction

First, I will talk about GAN. It is a model architecture for training a generative model, and deep learning models are commonly used in this architecture. The GAN architecture was first described in Ian Goodfellow paper [1], it involves two sub-models: a generator model for generating new examples and a discriminator model for classifying whether generated examples are real, from the domain, or fake, generated by the generator model. During training, the generator is constantly trying to outsmart the discriminator by creating better and better fakes, while the discriminator is working to improve his detective skills and correctly classify real and fake images. The process reaches equilibrium when the generator generates perfect fakes that appear to have come directly from the training data, and the discriminator is left to guess whether the generator output is real or fake with a 50% confidence level. Let's start with the discriminator and define some notation that will be used throughout the tutorial. Assume x is data that represents an image, The discriminator network D(x) produces the (scalar) probability that x came from training data rather than the generator. Because we are dealing with images, the input to D(x) is a CHW image with dimensions of 3x64x64. When x comes from training data, D(x) should be high, and when x comes from the generator, D(x) should be low. D(x) can also be viewed as a classic binary classifier. Let z be a latent space vector sampled from a standard normal distribution for the generator's notation. The generator function G(z) is used to map the latent vector z to dataspace. The goal of GG is to estimate the distribution that the training data comes from $p_{data}$ it can generate fake samples from that estimated distribution $p_g$. As a result, D(G(z)) is the probability (scalar) that the generator G's output is a real image. D and G play a minimax game in which D tries to maximize the probability that it correctly classifies reals and fakes (logD(x)), while GG tries to minimize the probability that DD will predict its outputs are fake (log(1-D(G(z)) as described in Goodfellow's paper. The GAN loss function is described in the paper.

$$min_G max_D V(D,G) = E_{x\sim p_{data}(x)}[logD(x)] + E_{z\sim p_z(z)}[\log(1-D(G(z)))] \quad (1)$$

In theory, the discriminator guesses whether the inputs are real or fake at random, and the solution to this minimax game is where $p_g = p_{data}$. However, the convergence theory of GANs is still under research, and models do not always train to this level in practice.

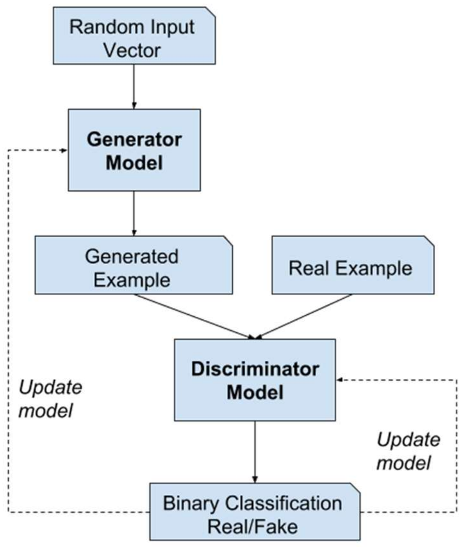


Figure 1. Example of the Generative Adversarial Network Model Architecture

Second, the reason for generating anime faces because with the passage of time, anime has grown in popularity. However, mastering the art of drawing necessitates a significant amount of effort before we can create our own characters. To fill this void, the automatic anime character generation allows users to create custom characters without requiring professional expertise.

In this project, I use Anime Face dataset that consisting of 36.7k high-quality anime faces for training my adversarial networks.

## 2. Methodology

### 2.1. Visualize the input images

To save time, initially use the random library to randomly generate 25 numbers from 0-36739 (name of the anime face image in the dataset), then visualize those images with the OpenCV library.

### 2.2. Pre-process and Load dataset

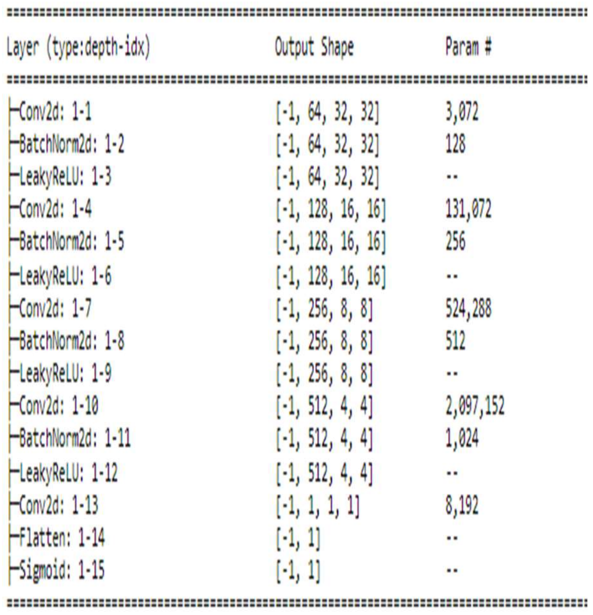
Check my device and move data to device (using cuda) for accelerate training process.

Both the generator and discriminator are trained with stochastic gradient descent with a modest batch size of 128 images.

Besides resize, center-crop and normalize by [(0.5, 0.5, 0.5), (0.5, 0.5, 0.5)] also try other data augmentation methods: Random Horizontal Flip and Random Adjust Sharpness.

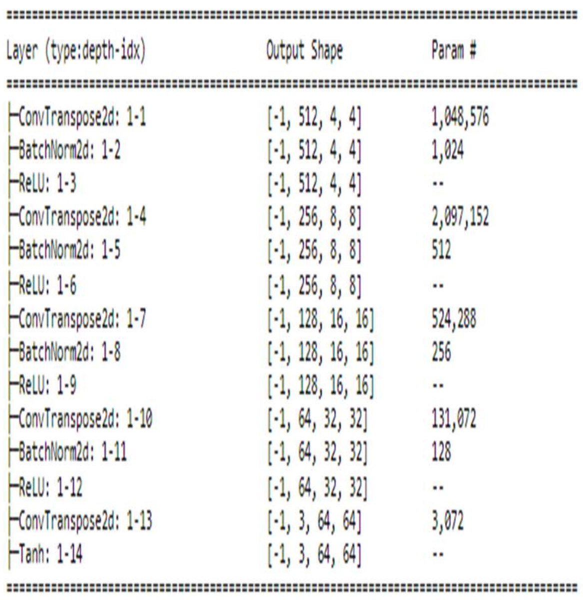
### 2.3. Define basic GAN, GAN with weights

Create a simple deep convolutional GAN, the data discriminator and generator have a structure as shown in Figures 2, 3, respectively.



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Conv2d: 1-1                            [-1, 64, 32, 32]          3,072
├─BatchNorm2d: 1-2                       [-1, 64, 32, 32]          128
├─LeakyReLU: 1-3                         [-1, 64, 32, 32]          --
├─Conv2d: 1-4                            [-1, 128, 16, 16]         131,072
├─BatchNorm2d: 1-5                       [-1, 128, 16, 16]         256
├─LeakyReLU: 1-6                         [-1, 128, 16, 16]         --
├─Conv2d: 1-7                            [-1, 256, 8, 8]           524,288
├─BatchNorm2d: 1-8                       [-1, 256, 8, 8]           512
├─LeakyReLU: 1-9                         [-1, 256, 8, 8]           --
├─Conv2d: 1-10                           [-1, 512, 4, 4]           2,097,152
├─BatchNorm2d: 1-11                      [-1, 512, 4, 4]           1,024
├─LeakyReLU: 1-12                        [-1, 512, 4, 4]           --
├─Conv2d: 1-13                           [-1, 1, 1, 1]             8,192
├─Flatten: 1-14                          [-1, 1]                   --
├─Sigmoid: 1-15                          [-1, 1]                   --
==========================================================================================
```

Figure 2. Structure of Discriminator



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─ConvTranspose2d: 1-1                   [-1, 512, 4, 4]           1,048,576
├─BatchNorm2d: 1-2                       [-1, 512, 4, 4]           1,024
├─ReLU: 1-3                              [-1, 512, 4, 4]           --
├─ConvTranspose2d: 1-4                   [-1, 256, 8, 8]           2,097,152
├─BatchNorm2d: 1-5                       [-1, 256, 8, 8]           512
├─ReLU: 1-6                              [-1, 256, 8, 8]           --
├─ConvTranspose2d: 1-7                   [-1, 128, 16, 16]         524,288
├─BatchNorm2d: 1-8                       [-1, 128, 16, 16]         256
├─ReLU: 1-9                              [-1, 128, 16, 16]         --
├─ConvTranspose2d: 1-10                  [-1, 64, 32, 32]          131,072
├─BatchNorm2d: 1-11                      [-1, 64, 32, 32]          128
├─ReLU: 1-12                             [-1, 64, 32, 32]          --
├─ConvTranspose2d: 1-13                  [-1, 3, 64, 64]           3,072
├─Tanh: 1-14                             [-1, 3, 64, 64]           --
==========================================================================================
```

Figure 2. Structure of Generator

I also custom weights initialization called on Generator, Discriminator (GAN-weights) and compare its performance with GAN.

### 2.4. Loss function and Optimizers

With D and G setup, we can specify how they learn through the loss functions and optimizers. We will use the Binary Cross Entropy loss (BCELoss) function which is defined in PyTorch as:

$$l(x,y) = L = \{l_1, \dots, l_N\}^T, l_N = -y_n \cdot log x_n + (1 - y_n) \cdot \log(1 - x_n) \qquad (2)$$

Define our real label as 1 and the fake label as 0. These labels will be used to calculate the losses of D and G, and the original GAN paper followed this convention. Finally, we created two optimizers, one for D and the other for G. Both are Adam optimizers with a learning rate of 0.0002 and $\beta_1 = 0.5$, as specified in the DCGAN paper [2]. we will generate a fixed batch of latent vectors drawn from a Gaussian distribution (i.e., fixed noise) to keep track of the generator's learning progress and periodically feed this fixed noise into G in the training loop, and over the iterations to see images emerge from the noise.

### 2.5. Training

The purpose of Discriminator training is to maximize the probability of correctly classifying a given input as real or fake.

$$maximize \left[logD(x) + \log(1 - D(G(z)))\right] \qquad (3)$$

First, we will construct a batch of real samples from the training set, forward pass-through D, calculate the loss (log(D(x)), then calculate the gradients in a backward pass.

Secondly, we will construct a batch of fake samples with the current generator, forward pass this batch through D, calculate the loss (log(1−D(G(z))), and accumulate the gradients with a backward pass. Now call a step of the Discriminator's optimizer with the gradients accumulated from both the all-real and all-fake batches.

We want to train the Generator by minimizing (log(1−D(G(z))) to generate better fakes. As previously stated, Goodfellow demonstrated that this does not provide sufficient gradients, particularly early in the learning process. As a workaround, we would like to maximize log(D(G(z))). We achieve this in the code by using the Discriminator to classify the Generator output, computing G's loss in a backward pass using real labels as GT and G's gradients, and finally updating G's parameters with an optimizer step.

## 3. Results

### 3.1. Visualize the input images

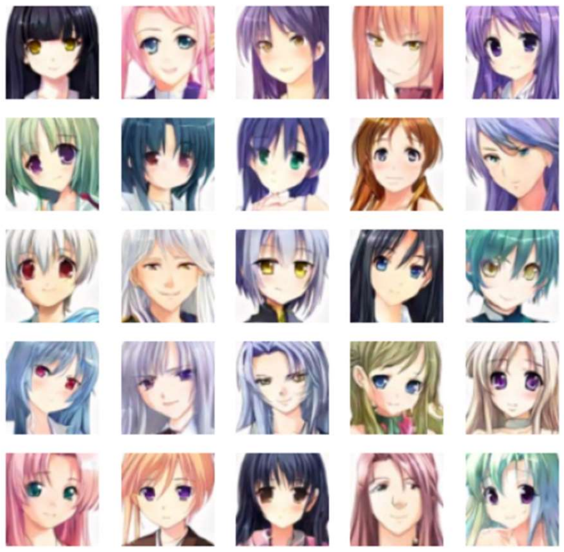


Figure 4. 25 random images from dataset

### 3.2. Training on GAN with different batch size

Using default parameter and transformers (resize, center-crop and normalize), I test 64, 128, 256 batch size on basic GAN and run on 40 epochs.

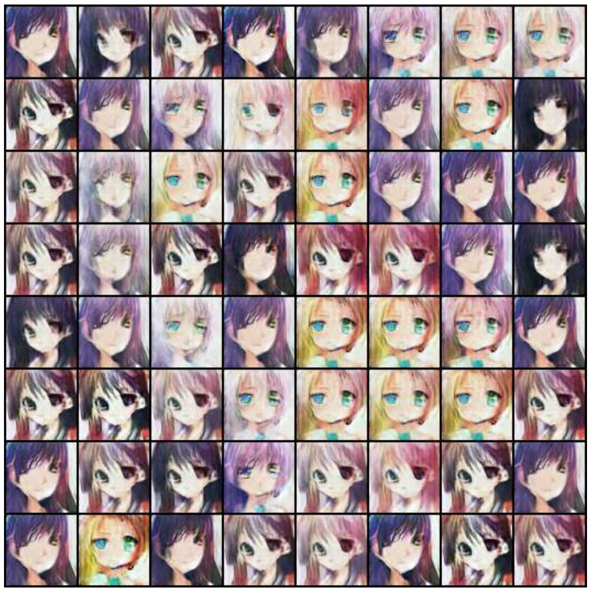


Figure 5. Generated images when batch size = 56, epoch = 40, GAN model

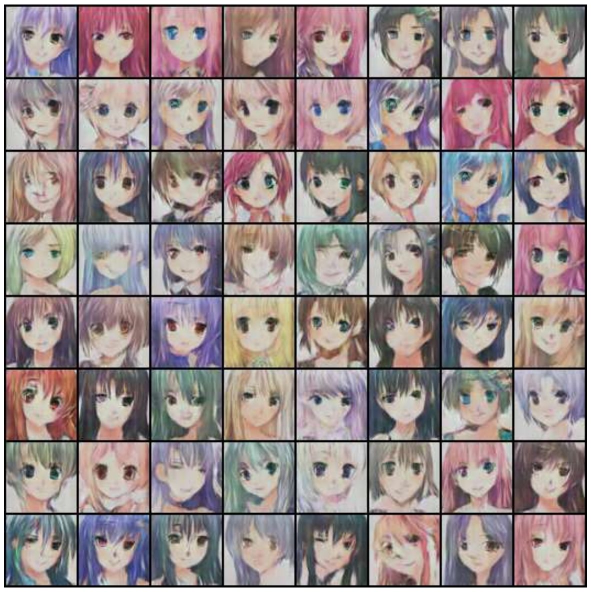


Figure 6. Generated images when batch size = 128, epoch = 40, GAN model

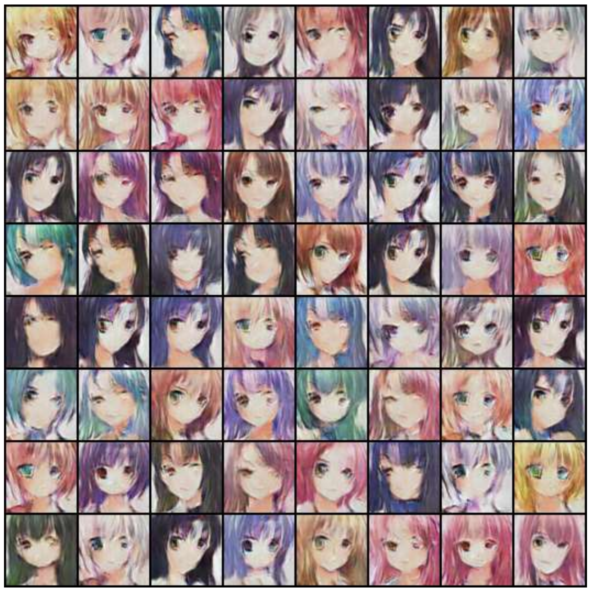


Figure 7. Generated images when batch size = 256, epoch = 40, GAN model

The generated image when using batch size 64 is quite similar and not very realistic, the training time is more than 2 minutes per epoch. When batch size = 256 does not reduce the training time (about 2 minutes 30 seconds per epoch) and the generated image is not good either. So, using 128 batch size is the most optimal, not only the generated images are diverse and quite good in all cases, but the training time is also the least under 2 minutes per epoch.

## 3.3. Generated images by GAN and GAN-weights

Using default parameter and transformers, batch size = 128, epochs = 40. Training data on GAN and GAN-weights.

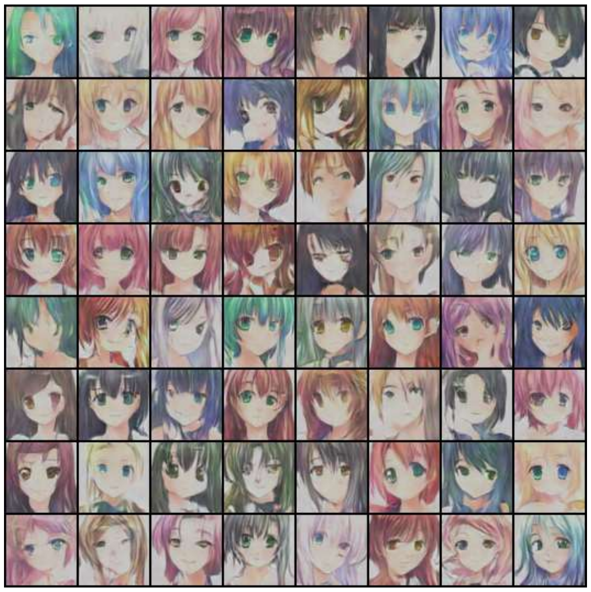


Figure 8. Generated images when batch size = 128, epoch = 40, GAN-weights model

The image generated by GAN-weights (Figure 8) looks smoother and more like an anime face, while GAN's generated image (Figure 6) is still rough and the facial parts are not harmonious. So, I will choose the GAN-weights model to test other augmentation methods for data.

## 3.4. Augmentation methods

Figures 6, 8 are the images generated after using the default augmentation methods: resize, center-crop and normalize for data for GAN, GAN-weights model. In addition, I also tried default augmentation methods combined with: Random Horizontal Flip or Random Adjust Sharpness using GAN-weights model.

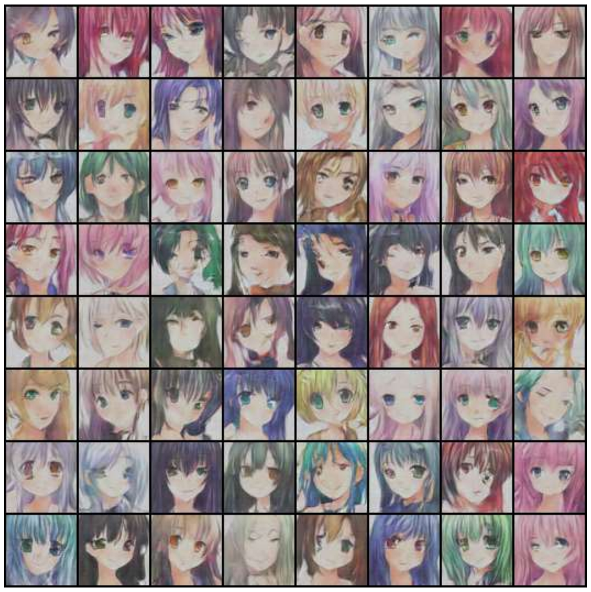


Figure 9. Generated images when using resize, center-crop, normalize and random horizontal flip for data on GAN-weights

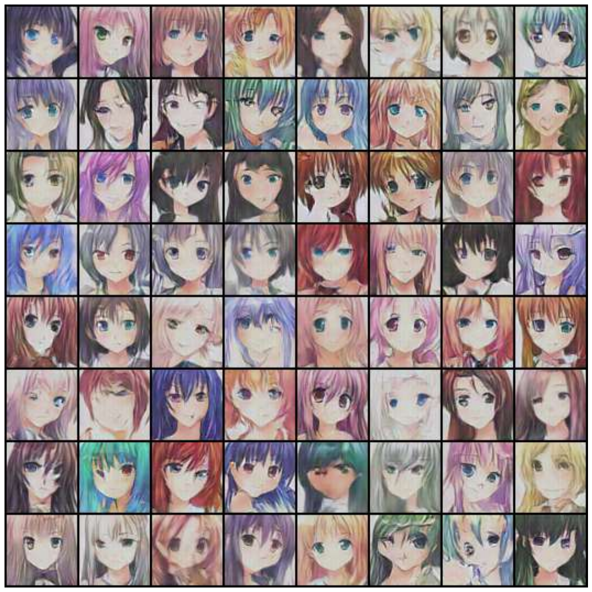


Figure 10. Generated images when using resize, center-crop, normalize and random adjust sharpness for data on GAN-weights

Default augmentation methods combined with: Random Horizontal Flip or Random Adjust Sharpness does not improve the generated image so temporarily the parameter that makes the GAN-weights model run best is batch size = 128, epoch = 40, latent size = 128, learning rate = 0.0002, $\beta_1$ = 0.5, use Adam optimizer and transform data by resize, center crop, normalize.

### 3.5. Training losses

Figure 11 shows the training losses of Discriminator and Generator of GAN-weights model with the parameter mentioned above.

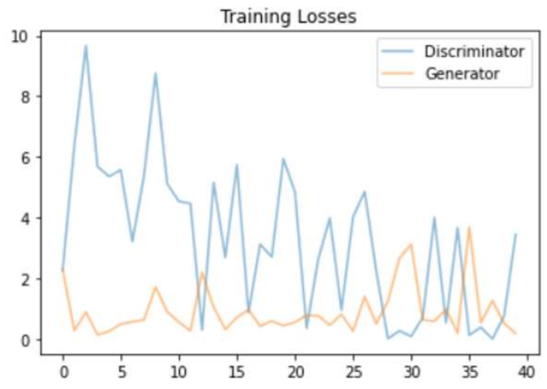


Figure 10. Training Losses

## 4. Conclusions

GAN-weights model performed better than GAN model with selected batch size, augmentation methods, and other parameters.

Augmentation method: Random Horizontal Flip or Random Adjust Sharpness does not improve the generated image. I think because anime face is drawing so sharpness adjustment doesn't work, and anime face structure sometimes doesn't match reality and drawing angle is not suitable to flip horizontally.

Based on Figure 10 we see that Generator losses are usually smaller than Discriminator losses, Generator losses are gradually decreasing over 40 epochs.

To make the generated image more realistic, I think it is necessary to clean the dataset because I found that there are many anime face images in the dataset that have errors that affect the training model. Anime face is a drawing created by human imagination, so it is very diverse in hair, eye, and skin color, so I think it is necessary to label the data based on those characteristics. Using a complex model (Style-GAN [3], DRAGAN [4]) and increasing the number of epochs is essential to learn more important features.

## References

[1] Ian J. Goodfellow, "Generative Adversarial Networks", [stat.ML] 10 Jun 2014.
[2] Alec Radford, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks", ICLR 2016.
[3] Tero Karras, "A Style-Based Generator Architecture for Generative Adversarial Networks", [cs.NE] 29 Mar 2019.
[4] Yanghua Jin, "Towards the Automatic Anime Characters Creationwith Generative Adversarial Networks", Comiket 92 (Summer 2017).