CSC412 Final Project Report: Using DC GAN to Generate Anime Faces

Yibing Sheng (shengyib), Chen Liang (liangc40)

Introduction

With Gen Z's ever-escalating passion for anime and ACG-related products, anime faces gradually have broad uses, such as avatars for various games and social media platforms. As these amine faces are among the top avatar choices for youngsters, the future business perspectives for anime face generators are promising. When I learned generative adversarial networks (GAN) [1] models from CSC412, I knew that GAN models could generate data with similar characteristics, especially for image datasets. My teammate and I figured we might create an anime face generator of our own by using a GAN model. We expect both the generator and the discriminator of our model could learn from the dataset. With the training process going on, the generator could generate better samples while the discriminator becomes more skillful at flagging samples. We expect to see our generator and discriminator contest with each other, and the discriminator's model's error rate increases, or in other words, the generated anime face images are hard to discriminate from the actual ones from the training dataset, and an acceptable accuracy has been achieved. My teammate and I learned various GAN models online, such as the context-guided GAN model (CGGAN)[2], the auxiliary classifier GAN model (ACGAN)[3], and the deep convolutional GAN model (DCGAN)[4], and we chose CGGAN as our core model for anime face generator. Reasons for choosing CDGAN are that compared with the vanilla GAN model, the deep convolutional one replaces the generator's fully connected networks with fractionally-strided convolutional layers, making the model more stable and reduces the converging speed. Convolutional neural nets find spatial correlations and are more fitting for image and video data. In contrast, the vanilla GAN could be applied to broader domains. Besides, in the deep convolutional model, all layers except the output layer of the generator and the input layer of the discriminator use batch normalization, which effectively prevents the generator from converging all samples to the same point. Hence, we decided to choose DCGAN as our core model to implement.

Methods

Data Collection: We collected our data from the Kaggle dataset. The dataset consists of 21151 64 px * 64 px anime face images, and all images have similar face position and artistic style. We read in all the images by using PIL and normalize the image data from 0 to 1. [4]

Model Structure:
Here I listed some details for the model structures:
1. Replace all max-pooling with convolutional strides.
2. Use transposed convolution for upsampling.
3. Use batch norms for both the generator and the discriminator.

4. Use ReLu Activation in the generator for all layers except for the output layer.
5. Use Leaky ReLu Activation in the discriminator for all layers except for the flattening layer and the output layer.
6. The output layers for both the generator and the discriminator both use the sigmoid activation function.

Below I showed the model details and architectures for the generator and the discriminator.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	2432
leaky_re_lu (LeakyReLU)	(None, 32, 32, 32)	0
dropout (Dropout)	(None, 32, 32, 32)	0
conv2d_1 (Conv2D)	(None, 16, 16, 64)	51264
zero_padding2d (ZeroPadding2	(None, 17, 17, 64)	0
batch_normalization (BatchNo	(None, 17, 17, 64)	256
leaky_re_lu_1 (LeakyReLU)	(None, 17, 17, 64)	0
dropout_1 (Dropout)	(None, 17, 17, 64)	0
conv2d_2 (Conv2D)	(None, 9, 9, 128)	204928
batch_normalization_1 (Batch	(None, 9, 9, 128)	512
leaky_re_lu_2 (LeakyReLU)	(None, 9, 9, 128)	0
dropout_2 (Dropout)	(None, 9, 9, 128)	0
conv2d_3 (Conv2D)	(None, 5, 5, 256)	819456
batch_normalization_2 (Batch	(None, 5, 5, 256)	1024
leaky_re_lu_3 (LeakyReLU)	(None, 5, 5, 256)	0
dropout_3 (Dropout)	(None, 5, 5, 256)	0
flatten (Flatten)	(None, 6400)	0
dense (Dense)	(None, 1)	6401

Total params: 1,086,273
Trainable params: 1,085,377
Non-trainable params: 896

figure 1: model detail for generator

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 16384)	1654784
reshape (Reshape)	(None, 8, 8, 256)	0
batch_normalization_3 (Batch	(None, 8, 8, 256)	1024
activation (Activation)	(None, 8, 8, 256)	0
up_sampling2d (UpSampling2D)	(None, 16, 16, 256)	0
conv2d_4 (Conv2D)	(None, 16, 16, 128)	819328
batch_normalization_4 (Batch	(None, 16, 16, 128)	512
activation_1 (Activation)	(None, 16, 16, 128)	0
up_sampling2d_1 (UpSampling2	(None, 32, 32, 128)	0
conv2d_5 (Conv2D)	(None, 32, 32, 64)	204864
batch_normalization_5 (Batch	(None, 32, 32, 64)	256
activation_2 (Activation)	(None, 32, 32, 64)	0
up_sampling2d_2 (UpSampling2	(None, 64, 64, 64)	0
conv2d_6 (Conv2D)	(None, 64, 64, 32)	51232
batch_normalization_6 (Batch	(None, 64, 64, 32)	128
activation_3 (Activation)	(None, 64, 64, 32)	0
conv2d_7 (Conv2D)	(None, 64, 64, 3)	2403
activation_4 (Activation)	(None, 64, 64, 3)	0

Total params: 2,734,531
Trainable params: 2,733,571
Non-trainable params: 960

figure 2: model detail for discriminator

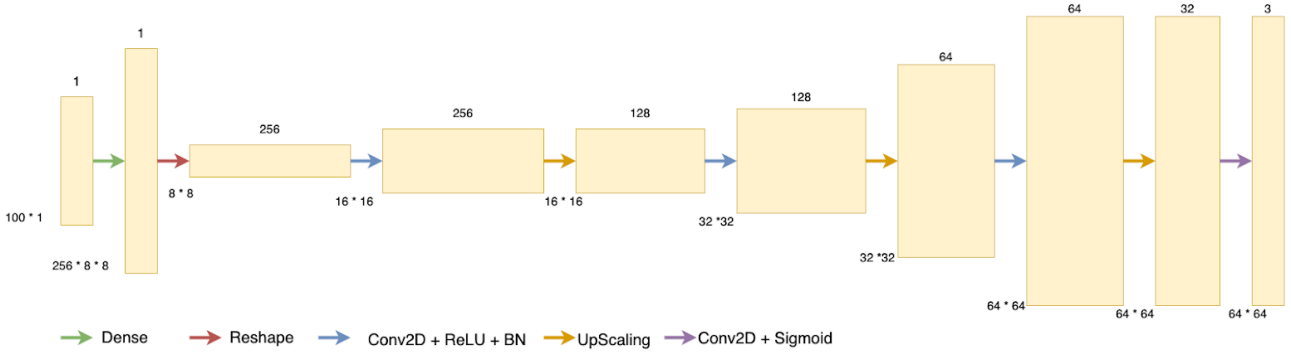

figure 3: architecture for the generator

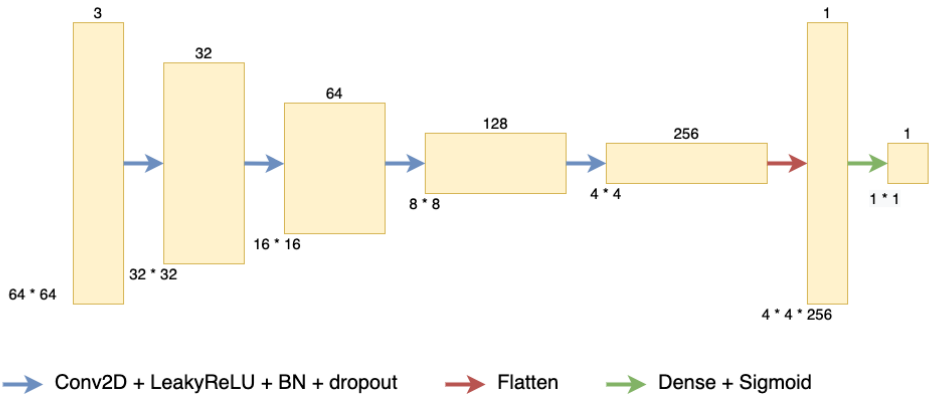

figure 4: architecture for the discriminator

Training Method:

training loop:

In each epoch, we train both the discriminator and the generator.
For the discriminator training, we first choose a set of random images from the training dataset, then choose random noise to generate a new batch of new images. Then train the discriminator by learning the random images from the training dataset and the generated images as zeros.

```
# ---------------------
#  Train Discriminator
# ---------------------
noise = np.random.normal(0, 1, (batch_size, self.latent_dim))
gen_imgs = self.generator.predict(noise)

for j in range(dis):
    # Select a random half of images
    idx = np.random.randint(0, X_train.shape[0], batch_size)
    imgs = X_train[idx] # (128, 28, 28, 1)  vs (128, )
    # Sample noise and generate a batch of new images

    # Train the discriminator (real classified as ones and generated as zeros)
    d_loss_real = self.discriminator.train_on_batch(imgs, valid)
    d_loss_fake = self.discriminator.train_on_batch(gen_imgs, fake)
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
```

figure 5: code for training the discriminator

And for the training for the generator, we use the combined model and want the discriminator to misclassify the ones created by the generator as real.

```
# ---------------------
#  Train Generator
# ---------------------

# Train the generator (wants discriminator to mistake images as real)
for j in range(gen):
    # g_loss = self.combined.train_on_batch(noise, valid)
    noise = np.random.normal(0, 1, (batch_size, self.latent_dim))
    g_loss = self.combined.train_on_batch(noise, valid)

# Plot the progress
print ("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" % (epoch, d_loss[0], 100*d_loss[1], g_loss))
acc = d_loss[1]
# If at save interval => save generated image samples
if epoch % save_interval == 0:
    self.save_imgs(epoch)
```

figure 6: code for training the generator

loss function, optimizer and hyperparameter:

We use binary cross entropy as the loss function and the Adam Optimizer with the learning rate as 0.0002 and the beta_1 as 0.5

Experiments and Discussion

Dynamic Training Strategy

During the training process, we develop an idea that can dynamically allocate the training weight to the generator and discriminator. Initially, we define the training weight as equal to the discriminator's accuracy. Our target is to train the discriminator more times when the accuracy of the last epoch is low and to train the generator more times when the accuracy of the last epoch is high. The logic behind this dynamic training process is that when the discriminator's accuracy is high, training the generator few more times will increase the chance for the discriminator to mistaken the generated images as real; and when the discriminator's accuracy is low, training the discriminator will make it more skillful at distinguishing the real ones and generated ones. To achieve this, we set the training times for the generator as *int(max(10 * (1 - accuracy), 3))* times, and the training times for the discriminator as *int(max(10 *accuracy, 1))* times. Besides, we set the lower bound number of training in each epoch as 3 to the generator as we hope even if the accuracy is high, we still need to train a reasonable amount of times to cheat the discriminator. According to this strategy, if the discriminator was trained and performed well, the accuracy will be high and cause the discriminator to be trained fewer times than the generator. Generally, the generator improves much slower than the discriminator does.

We design such a training strategy because it takes too long to train the generator to cheat the discriminator significantly. Suppose we always put more effort into training the generator. After a certain number of epochs, the generator will ultimately beat the discriminator, and the discriminator will ultimately fail, and considerably high accuracy for the generated images could be achieved.

In the following part, we will demonstrate the comparison between the general training strategy (one for generator and one for discriminator for each epoch) and the dynamic one elaborately.

General Training Strategy vs Dynamic Training Strategy

0 - 100 epochs

The general training strategy trains the discriminator and the generator once in each epoch.
We set 20 epochs as the save window, which means at the end of every 20 epochs, we use the generator to generate a picture and save it. First, we will compare the first six pictures generated in the first 100 epochs.

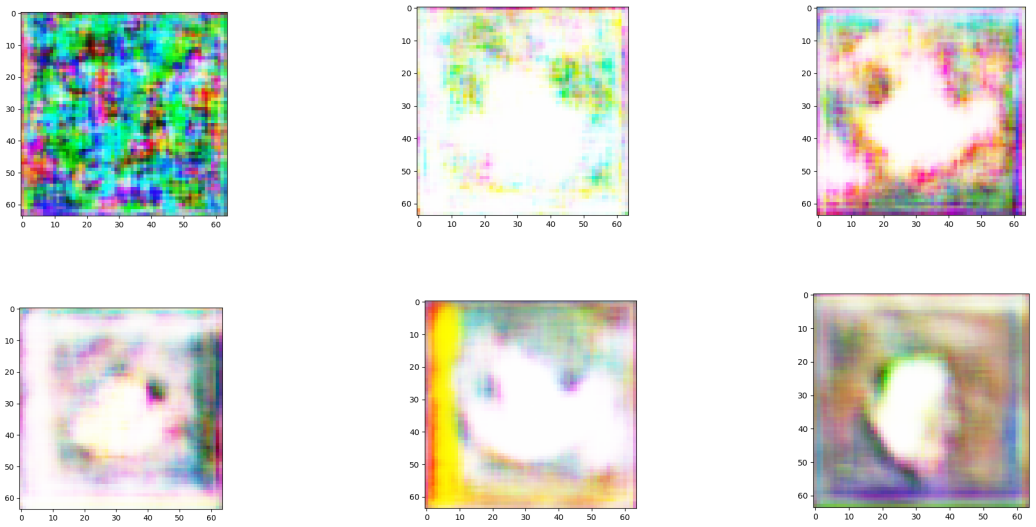

Figure 7. Faces generated using the general training strategy at 0-100 epochs

These six pictures are generated under the general training strategy. The first picture was generated at the end of the first epoch, and the last picture is generated at the end of the 100-the epoch. As we can see, the shape of the face is vaguely sketched.

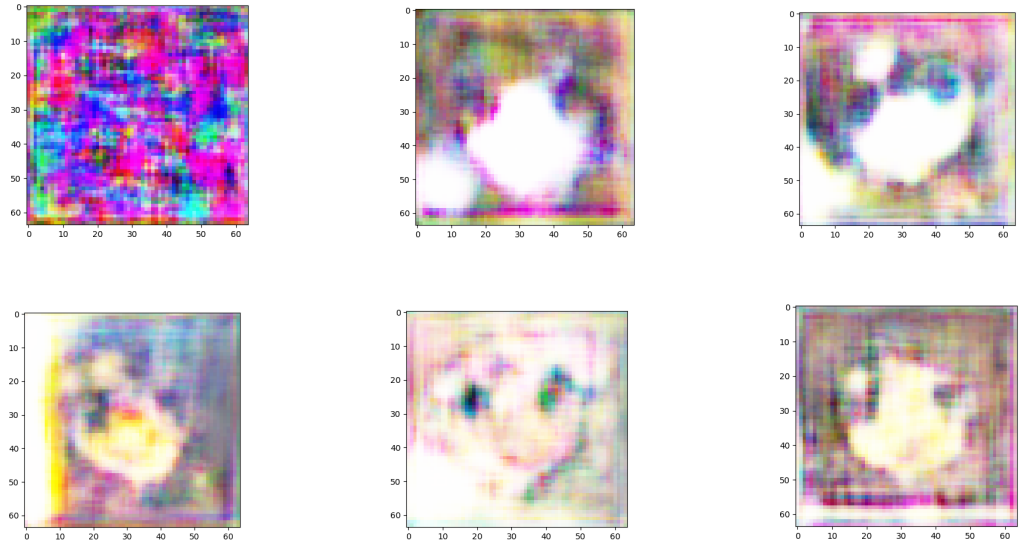

Figure 8. Faces generated using the dynamic training strategy at 0-100 epochs

These six pictures are generated under our dynamic training strategy illustrated above. We can notice that using our designed training strategy can significantly improve the DCGAN in the first 100 epochs

from these generated pictures. The dynamic training model could better capture facial features such as eyes, and the colorization is more vivid.

700-800 epochs

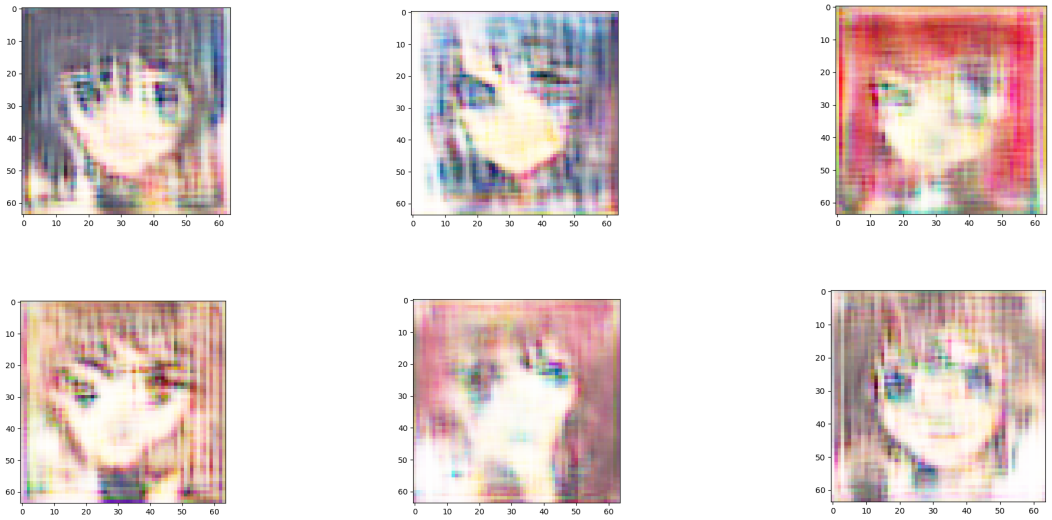

Figure 9. Faces generated using the general training strategy at 700 - 800 epochs

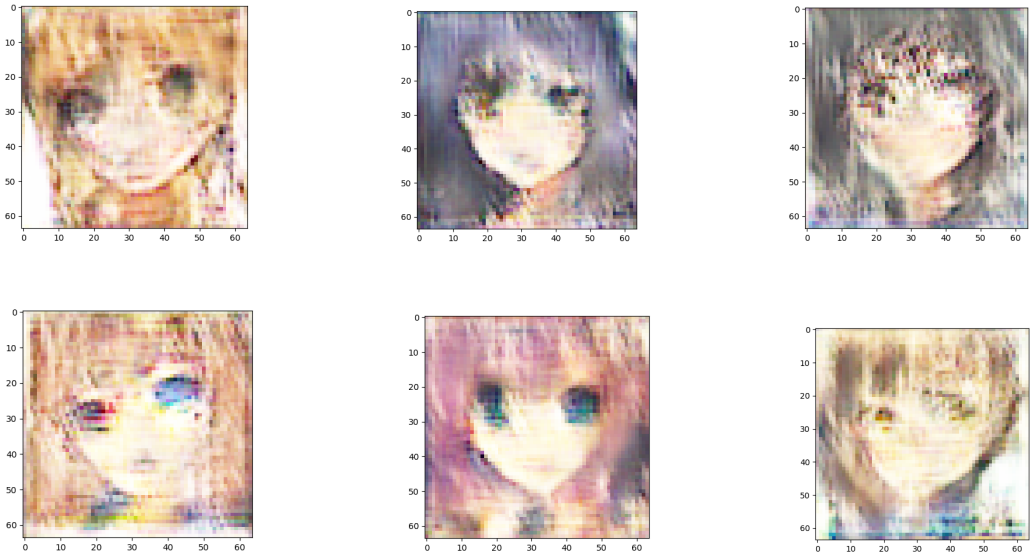

Figure 10. Faces generated using the general training strategy at 700 - 800 epochs

Figure 9 are faces generated under general training strategy and figure 10 are faces generated under our designed training strategy. The qualities do not vary a lot. However, we can still notice that the boundaries in figure 10 are clearer, which can show that our training strategy can better capture the parts with deeper color. Since the color of hair and eyes are better demonstrated, then the boundary is better shown.

1300-1400 epochs

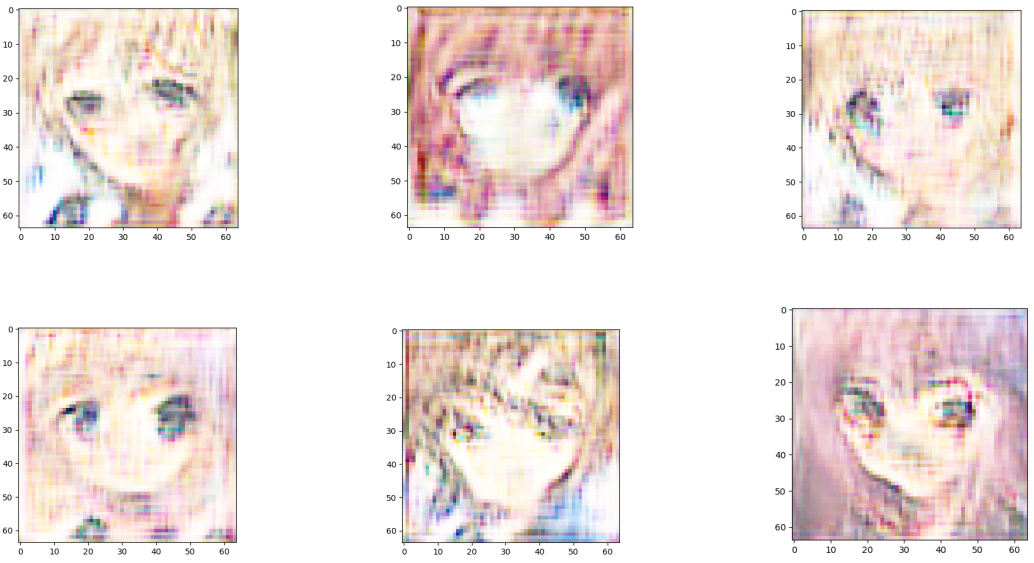

Figure 11. Faces generated using the general training strategy at 1300 - 1400 epochs

Figure 12. Faces generated using the general training strategy at 1300 - 1400 epochs

As we can see, the qualities of generated pictures compared to those at 700-800 epochs do not improve too much. The discriminator's loss and the generator's loss are bouncing around 0.2 after the 300th epoch. The discriminator's accuracy fluctuates between the range of 43% to 68% after the 300th epoch. In other words, the dcgan is converged no matter which way we train it. My explanation is that the model has reached its limitation, and there is no way to improve it unless we improve the model.

In summary, using our training strategy can make the model converge in a short time. We used to fine-tune other hyperparameters during the training, but we ran out of time due to computing limitations. Therefore, we only test two different training strategies, and for each, we only train 1400 epochs.

Conclusion

In conclusion, we successfully finished the code implementation for the DCGAN model, and by using DCGAN we could generate anime faces with relatively clear facial features and the artistic style look pretty similar to the training data. We also devised a dynamic training process to ensure the generator and the discriminator always always compete with each other, in order to increase the model accuracy. We failed to find the optimal hyperparameters for the training process as the training takes an excessively long time, even for a few days. We will replace the DCGAN model with other GAN modes such as CGGAN, StyleGAN, StyleGAN 2 etc, to see which model could achieve the best training result both in terms of training speed and facial features capturing. With our future endeavours and improvements, when our anime face generator is maturely developed, we will post our code open source online and we hope it could generate anime faces which youngsters find useful.

References

[1] I.J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, *Generative Adversarial Nets*, NIPS Proceedings (2014). https://arxiv.org/abs/1406.2661 Accessed Apr 2021.

[2] Zhaorun Zhou, Zhenghao Shi, Mingtao Guo, Yaning Feng, Minghua Zhao, *CGGAN: A Context Guided Generative Adversarial Network For Single Image Dehazing*, (2016). https://arxiv.org/abs/2005.13884 Accessed Apr 2021.

[3] Augustus Odena, Christopher Olah, Jonathon Shlens, *Conditional Image Synthesis With Auxiliary Classifier GANs*, (2019). https://arxiv.org/abs/1610.09585. Accessed Apr 2021.

[4] A. Radford, L. Metz, S. Chintala, *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, ICLR (2016). https://arxiv.org/abs/1511.06434 Accessed Apr 2021.

[5] Soumik Rakshit, *21551 Anime Face images sctaped from www.getchu.com,* (2019). https://www.kaggle.com/soumikrakshit/anime-faces Accessed Apr 2021.