Final Project Individual Report

1 Introduction

In this project, we applied optimization techniques on DCGAN by modifying model architecture to perform better face generation on Anime Faces data. We analyze our model by looking at discriminator loss, generator loss, and the quality of the resulting image.

DCGAN (Deep Convolutional Generative Adversarial Network) is a type of GAN used for image generation tasks, which uses CNNs for both the generator and discriminator networks. The generator takes a noise vector and produces an image, while the discriminator predicts whether an image is real or fake. During training, they learn from each other in an adversarial manner. To make DCGAN more stable, architectural guidelines such as batch normalization and ReLU activation functions are followed. DCGAN has been used in various applications, such as style transfer and data augmentation, to generate high-quality images.

2 Description of my individual work

2.1 Creating the Base Model

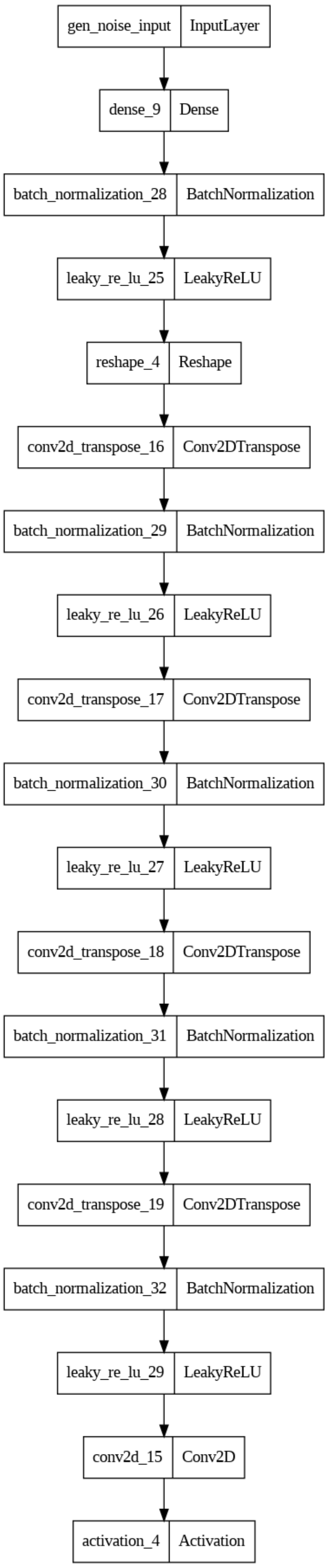
I created a class called GAN that contains the entire Deep Learning model with preset parameters. This enables us to work on the model without encountering other issues, and any necessary modifications can be made within the class without fear of generating errors.
The GAN class includes the generator, discriminator, and train step code. Building of the generator and discriminator is done in the init() method, eliminating the need for manual building. To train, simply instantiate the class, making it very user-friendly and simple to train.

2.2 Graph Plotting and Video Writer

I developed a flexible class named Utils that consists of functions for data preprocessing, graph plotting, and image generation. These functions can be utilized for a range of data sizes and types, making it highly versatile, making it usable for other projects as well
I developed a video writer Class that takes an image as input and writes it to a video file for a specified number of frames.

2.3 Base Model

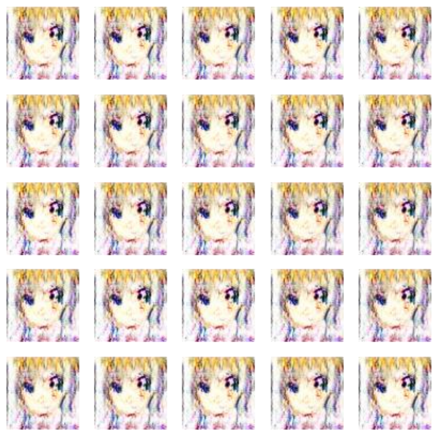
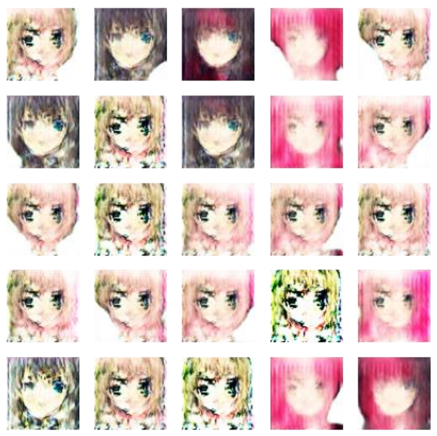
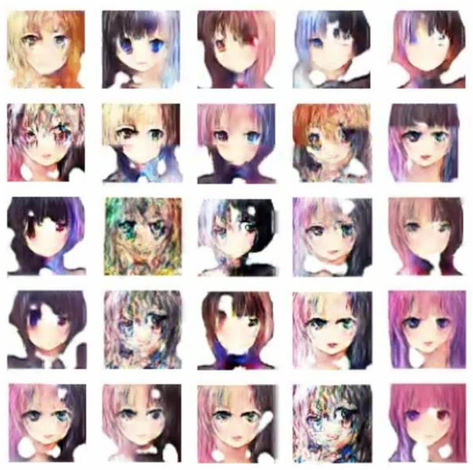
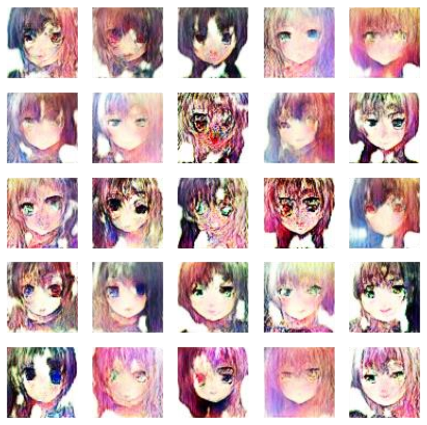
The architecture of the base model was improved by organizing it better and is shown below.


gen_noise_input | InputLayer
dense_9 | Dense
batch_normalization_28 | BatchNormalization
leaky_re_lu_25 | LeakyReLU
reshape_4 | Reshape
conv2d_transpose_16 | Conv2DTranspose
batch_normalization_29 | BatchNormalization
leaky_re_lu_26 | LeakyReLU
conv2d_transpose_17 | Conv2DTranspose
batch_normalization_30 | BatchNormalization
leaky_re_lu_27 | LeakyReLU
conv2d_transpose_18 | Conv2DTranspose
batch_normalization_31 | BatchNormalization
leaky_re_lu_28 | LeakyReLU
conv2d_transpose_19 | Conv2DTranspose
batch_normalization_32 | BatchNormalization
leaky_re_lu_29 | LeakyReLU
conv2d_15 | Conv2D
activation_4 | Activation

3 Description of my individual work in Detail with Results

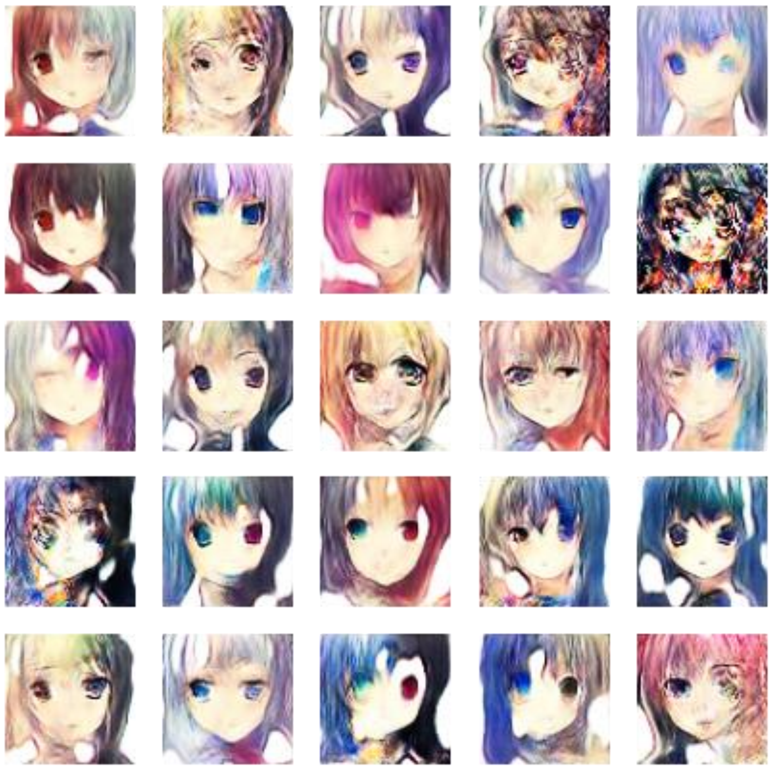
3.1 LeakyReLU Values

I primarily experimented with the LeakyReLU values. I tried 0.05, 0.1, 0.2, 0.3, 0.5. The results varied for each. I noticed that as I reduced the LeakyReLU, the GAN suffered from mode collapse, i.e. the GAN was only producing very limited number of samples. A value of 0.05 lead to a complete collapse producing only on image causing me to stop training early. A value of 0.1 had mode collapse but not as drastic as 0.05. A value 0.2 performed well. In my opinion 0.3 performed the best. A value of 0.5 performed good but the features in the images had sort of merged and were not looking very good. Below are images of LeakyRelu of value 0.05, 0.1, 0.2, 0.5 in order.

3.2 Number of Filters

The number of filters were 32 * 2^i for generator and 64* 2^i for discriminator, where I is the ith layer of discriminator or generator. This was changed to 64 * 2^i for generator and 128* 2^i for discriminator and seemed to improve performance. The convergence occurred much faster.



3.3 Kernel Size

Lowering the kernel size to 3 seemed to give better results overall, but the kernels failed to capture more features due to their small size and were not used.
Kernel size of 7 seemed to lead to a vanishing gradient in the first few layers of the Generator and gave white/gray images as output.

3.4 Progressively growing Kernel Size

Since the Kernel size of 3 was giving good results and a higher kernel size only caused vanishing gradient in the first few layers, this gave me the idea of increasing the kernel size with the layers. The size of the kernel would be (i*2)+1, where i is the ith layer. This seemed to give really good results at the beginning, but since it took a very long time to train, around 5 minutes per epoch, I did not go forward with it due to time constraints. This idea was inspired by ProGANs.

3.5 Loss Function

The loss function is one of the most important considerations for a GAN. I had read many paper (in references) to try to understand and create a custom loss function, but was not able to understand the mathematics as it was too complicated. I stuck to using BinaryCrossEntropy.

$$logloss = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{M} y_{ij}\log(p_{ij})$$

- N is the number of rows
- M is the number of classes

4 Summary and Conclusions

After experimenting a with a variety of parameters and training over 20 models, this is the conclusion I have come to:

- LeakyRelu is very important for the GAN to generate any meaningful results. Using only ReLU added a lot of noise to the GAN.
- Tanh worked for the output of the convolution layer in the generator. Sigmoid worked but generated not so good results. ReLU did not work.
- The value of LeakyReLU changes the output drastically. Lowering the alpha leads to mode collapse and increasing it leads to improperly capturing features
- Adding more filters generate better output but also increase the training time. Doubling the filters did give a better result, but I highly doubt that adding more filters will give a better result as these filter may contain the same information
- Progressive growing of kernel size could give better results but drastically increases training time
- 250 epochs was a good number as any more did not produce meaningful results even though the losses were decreasing (albeit at a much slower rate).
- The loss function by far is one of the most important aspects of GANs. This is extremely critical for results and realized this after reading different paper that placed emphasis of loss functions.
- Vanishing gradient is a real problem with GANs and should be taken into consideration while designing.
- Mode Collapse occurs quite frequently and measures must be taken to prevent it, for example, by adjusting LeakyReLU, using Dropouts, kernel sizes etc.

5 Code Percentage

The code on the internet is 208 lines. Around 50 were modified (considering each method parameter for conv block and deconv block is a different line) and 20 were added. Nothing could be changed in the train step which is most of the code. The percentage used from the internet is 72%.
If the restructuring of code is considered, then only around 20% of the code was used from the internet as the entire structure of the code was written from scratch.

6 Refrences

https://github.com/nikhilroxtomar/DCGAN-on-Anime-Faces/blob/master/gan.py
https://paperswithcode.com/method/dcgan
https://www.analyticsvidhya.com/blog/2021/07/deep-convolutional-generative-adversarial-network-dcgan-for-beginners/
https://arxiv.org/ftp/arxiv/papers/2112/2112.08196.pdf
https://arxiv.org/pdf/1704.03971.pdf