DCGAN for reproducing anime faces

Hannu Koivisto
University of Oulu
Pentti Kaiteran katu 1, 90570 Oulu
Hannu.Koivisto@student.oulu.fi

Abstract

This paper is a part of the final project of Deep Learning course. Throughout the course we have been fortunate enough to gain highly valuable knowledge from multiple different deep learning applications. This project work was done using given notebook frame and instructions by the course personnel. Our problem was to generate new anime face images from the given dataset using DCGAN. We implemented DCGAN following the instructions and then tried our own experiments to enhance our deep learning process and its efficiency. DCGAN was implemented successfully after multiple days of hard work and tweaking. In the end, results gotten were decent. Implementation concerned in this paper produced clear and easily recognizable images, but in the later epochs their color seemed to slowly dim.

1. Introduction

Success with discriminative deep learning models and their performance was originally way ahead of deep learning based generative models. Generative Adversial Networks (GANs) were made with the focus on improving abilities of generative deep learning models with the help of a discriminative model [1].

Basic architecture of a GAN consists of two models competing against each other. The discriminator tries to classify real and fake images correctly as the generator is continuously improving the fake images it produces to deceive the discriminator [1]. One of the most difficult things in designing a GAN is to stabilize its training. There remains a void to be filled regarding the theory of instability of GANs, but multiple different methods have been proposed to enhance their stability [2].

Use cases of GANs has been continuously growing as it is still a fresh innovation in the deep learning related research. Now days it is used for multiple different purposes e.g., 3D object generation, face detection, image processing, COVID-19 diagnosis, tumor segmentation, text transferring and traffic control [3].

New applications have been made possible by the evolution of GANs. Deep Convolutional GAN (DCGAN) introduced in 2015 was developed for improving the methods of unsupervised learning. It used deep convolutional layers in discriminator and generator in contrast to its predecessors. Achieving deeper networks demanded some specific architectural guidelines. Pooling layers were replaced with strided convolutions, batch normalization in both generator and discriminator, use of ReLU in generator, use of LeakyReLU in discriminator and no fully connected layers [4].

Our aim was to generate new anime faces from a given dataset by using generative adversial network. Data augmentation is a common use case of DCGAN and quite an important one. Many deep learning solutions demand huge amounts of data which data augmentation can help with. Implementing very basic DCGAN gave decent results and produced clear and sharp images of anime faces. Model considered in this

paper could still be improved multiple ways, but it would require more computational resources and knowledge.

2. Data

Data for the project was given by the course personnel. Dataset used is public and it can be found in Kaggle. Anime Face Dataset NTU-MLDS consists of 36740 images of different anime faces. Licensing of the dataset is unknown, but the owner's Kaggle account is *lunarwhite* [5]. Resolution of these images was 64x64. It is unknown if the images were drawn or not, but it looks like at least some of them have been produced by GAN (Fig.1).

Preprocessing of the dataset was conducted in the constructor of our dataset class. Transforms included resizing every image to 64x64 and cropping them at the center. Images were also transformed to tensors and normalized with standard deviation of 0.5 and range of -1 to 1. Transformations were done in parallel with the image loading using PyTorch's *ImageFolder*-method. Images were also shuffled while loading them to the network in batches of 64. This resulted in 575 iterations for one epoch.

3. Methods

All the code in this project work was done by using Python [6]. It is free and extremely popular platform for object-oriented programming with multiple open-source packages for specific use cases. Its versatility and user friendliness has made it the most popular programming language for now [7]

3.1. PyTorch

2016 released PyTorch is a popular deep learning library for Python. With the mindset of free software movement people have moved away from closed and strictly licensed software towards open-source Python ecosystem with multiple quality packages able handling complex computational problems. PyTorch is built following this trend with focus on usability without sacrificing speed and performance too much. API is designed to be intuitive and simple to use as complexities of deep learning are mainly hidden under the hood, but still available for more advanced tweaking. This makes PyTorch optimal deep learning environment for newcomers in this field and for more experienced AI scientist. [8]

PyTorch by Facebook is one of the two most popular deep learning frameworks with industrial standard TensorFlow released by Google [9]. Lately PyTorch has become even more popular than TensorFlow judged by the fraction of papers using PyTorch vs. TensorFlow and researchers migrating frameworks [10].

3.2. Architecture

Both networks proposed have been constructed by taking advantage of *nn.Sequential* method of PyTorch. It streamlines the construction of the models quite a lot as the defining of forward function gets almost automated. Using *nn.Sequential* also sets some architectural restrictions as it can only handle layers from Torch.nn. For example, this makes reshaping layers a bit more complicated. Both discriminator and generator consist of five main layers and neither uses bias in their convolutional nor transposed convolutional layers.

Discriminator used here reminds other discriminative network architectures. Instead of pooling layers it uses stride value of 2 in the first four convolutional layers. In the fifth layer stride is set to one. Five 2D convolution layers are initialized and batch normalization is used after each convolution layer. Kernel size is those convolutional layers is set to 4 and padding is set to 1 Only exception is the last layer where padding value is 0. LeakyReLU is selected for the role of activation function with slope value of 0.2 and calculations conducted in place. After the fifth convolutional layer there is no LeakyReLU or batch normalization to be seen. Sigmoid function is used as an activation function and lastly the output is flattened.

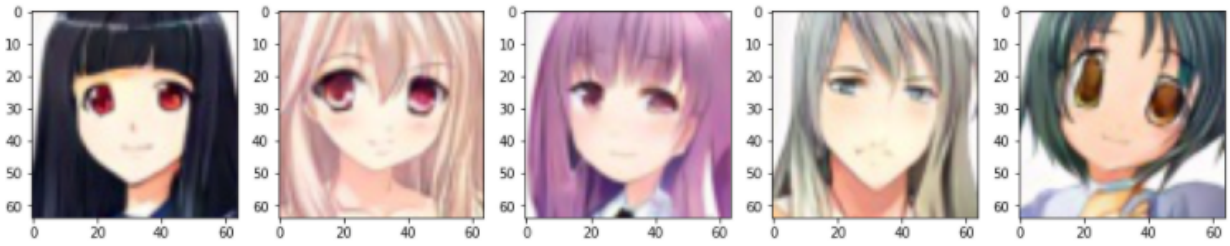


Figure 1. As we can see, the fifth image looks like it is artificially made. We can only guess the origin of it as the other images could be judged as real images of anime faces

Generator on the other hand is constructed around five 2D Transposed convolution layers. Kernel size is 4 in every layer and stride is set to 2 except the first layer where stride is set to 1. Padding value is also 1 in every layer except the first one where padding is set to 0. After every layer batch normalization is done and ReLU is used as an activation function afterwards with computation done in place. Only exception here is the last transposed convolution layer where there is no batch normalization or ReLU afterwards. Only Tanh is used before outputting the image generated.

3.3. Loss Function and Optimizer

Loss function implemented first was very basic cross entropy loss function from PyTorch's framework. Cross entropy loss was later replaced as it didn't produce good enough results and the selection was binary cross entropy loss (BCE Loss) used similarly by Zhang et al. [11]. In addition to the loss function, an accuracy score of discriminators ability of separating real images from fake images was also collected and used later in analysis.

Optimizer selected for this task was Adam, which is especially suited for large datasets by its efficient use of memory. Adam is also very efficient computationally and invariant to diagonal rescaling of the gradients. [12]

4. Implementation

In this section we are going to go through how the DCGAN was implemented and the difficulties faced during training and implementation. In the last section we go through experiments done during the implementation of the DCGAN in question.

4.1. How the DCGAN was implemented

Everything started by constructing the dataset class for importing dataset of anime faces. It was straightforward at first, but difficulties with applying image transforms changed methods of CV2-library to *ImageFolder* by Torchvision. It was able to handle transformations during the image importing.

Discriminator was implemented using *nn.Sequential* to wrap forward function of the network to a simplified package. LeakyReLU layers were first applied to discriminator with its default parameters. To enhance performance of the network, they were later adjusted by the example given in lecture 9. Slope was set to 0.2 and inplace was set to True. Otherwise, instructions were strictly followed.

Generator's functionality was also wrapped inside of a *nn.Sequential* module. Computations were also later done in place, as in LeakyReLU of the discriminator, after few unsuccessful tries with default parameters. Latent size was set to 128 as a default. During a short trial the untrained generator produced noisy color images as it should.

First loss function used was cross entropy loss, but it was shortly switched to a binary cross entropy loss (BCE loss) as it is more suitable for these purposes [11]. First optimizer implemented

was stochastic gradient descent. It was randomly selected to serve as a starting point, but soon deemed as substandard for this network. Instead, the Adam optimizer was introduced, and it managed to produce sufficient results. Learning rate was set to 0.0001 for discriminator and 0.0003 for the generator. Betas for both optimizers were 0.5 and 0.999. Ultimately a weight decay of 0.0005 was introduced to the optimizers.

Training loop with 40 epochs can be seen as the heart of our DCGAN. Implementing it correctly took easily the most time during the project work. First, the discriminator is trained to classify real and fake images with real images coming from our dataset and fake images produced by the generator. Afterwards the generator is trained to deceive the discriminator by giving fake labels to the loss function with discriminators output. Decision was made to use soft labels instead of hard ones to improve the performance of the generator.

Additional statistics and values are collected in the training loop to support the training and parameter tweaking of the model in question. Scores and losses from every epoch are plotted next to the images of the generator and their means are stored for later analysis. In the end, the images produced by our generator are saved after each epoch for analyzing generator's improvement. Mean losses from each epoch are saved for both discriminator and generator and plotted after training is over.

4.2. Challenges in implementation

First obstacle in the project work was the limitations of computing power of laptop used. It would have taken over 40hrs to train the network with the laptop available, so the focus turned on computational help offered by Google Colab. Eventually it was found out that Kaggle provided faster GPU support and more hours per week for their use. Eventually Kaggle was selected for the job, and it was able to complete training with 40 epochs in 1,5hrs.

Second impairing obstacle was the continuous images of nothing but colorful noise from the generator. Solution was found in the correct implementation of *detach*-method which was earlier used excessively. In the end those methods were replaced by setting *retain_graph*-parameter of discriminators backpropagation to 'True' (Fig. 2a).

After solving problems with noise images, the images produced by the generator were bluish and got very dark during the later epochs of the training loop. Solution was found by completely revamping the dataset class and setting *shuffle*-parameter in data loader to 'True' (Fig. 2b).

During the solving of these problems, many other tweaks and experiments were made to enhance the performance of the model.

4.3. Experiments

Using extra graphical presentations of the model's performance was critical in tweaking the parameters to the right direction. Generator's loss and both losses of the discriminator were plotted on the same graph. Also, the real and fake score of the discriminator were plotted into their own figure to see if they were interacting with each other in a suitable way (Fig. 3).

Optimizer values were tweaked quite much. Having beta values at their default ended up in oscillation of the losses. Parameters were then tweaked to find optimal beta values from 0.5 and 0.999. Learning rate was then adjusted separately for discriminator and generator. Discriminators learning rate was set to 0.0001 and generators to 0.0002 to maintain better stability longer during the training and preventing discriminator outperforming the generator too quickly.

Noisy labels were also introduced to the training loop for distracting the discriminator a bit more [13]. Noisy labels were also tried in the generator's training phase, but it had a negative effect on generators image quality. In the discriminator's training loop, it worked well and increased the stable time of the training.

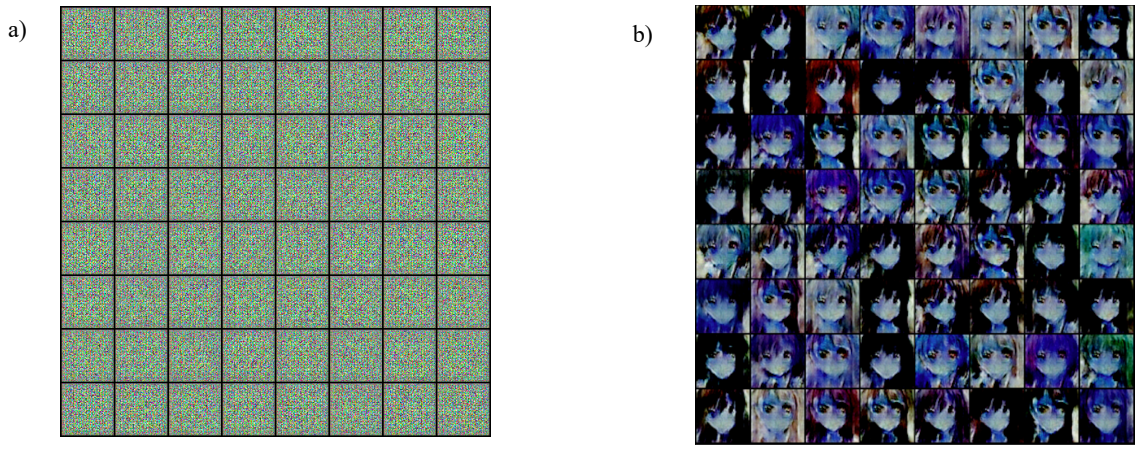


Figure 2. Images of complete noise with total lack of convergence (a). Bluish and dark images from the faulty model (b)

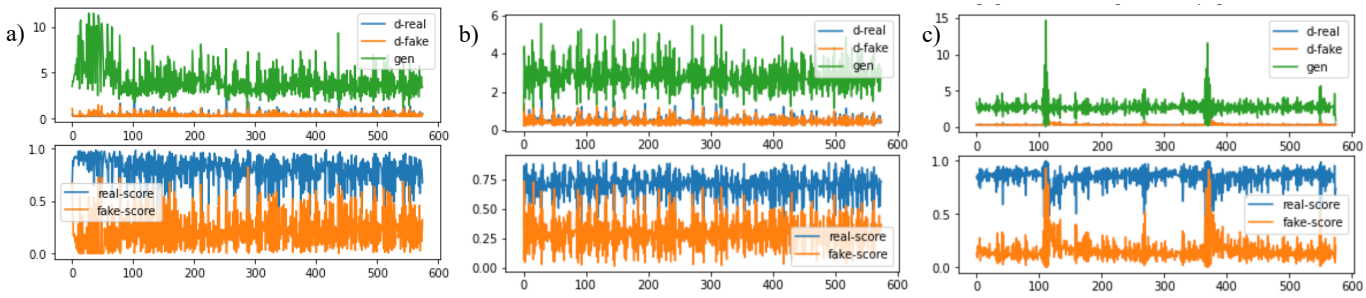


Figure 3. Evaluation parameters of second epoch of during training (a). Evaluation parameters of the 20th epoch during training(b) and the evaluation parameters of the 40th (last) epoch of the training (c) where we can clearly see the end of convergence.

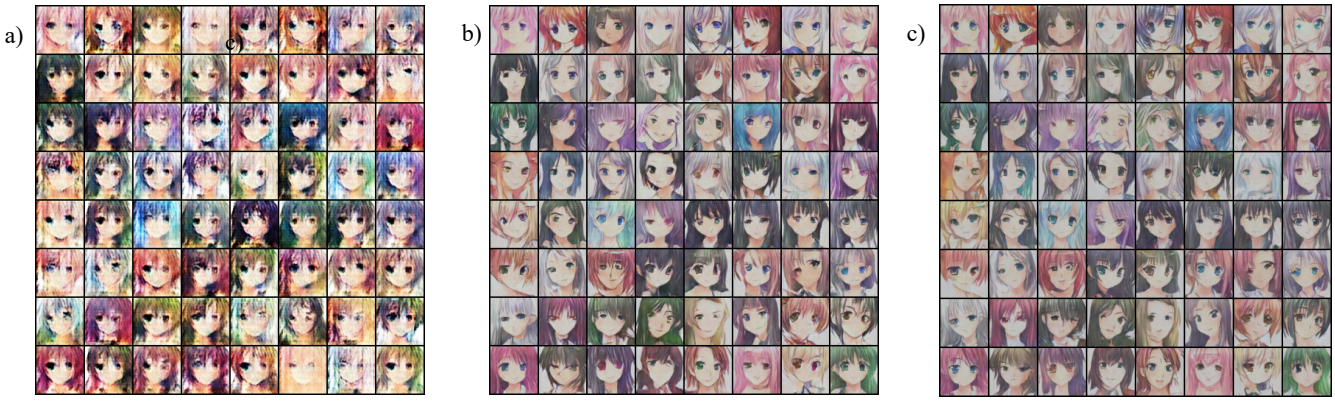


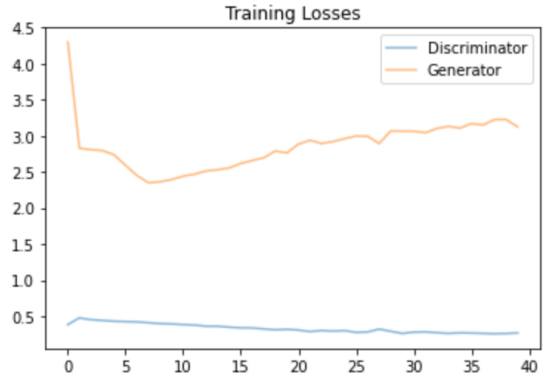
Figure 4. Generator produced images after second epoch (a). Generator produced images after 20th epoch (b) and the generator produced images after 40th epoch (c)

5. Results and Conclusion

Results obtained were decent for a first timer with GANs despite images produced getting dimmer towards the latter epochs. Some of the images in the latter epochs can be seen as convincing images of anime faces (Fig. 4.). Of course, the network doesn't achieve perfection, but results of this project can be seen promising, and at least the data gotten from performance metrics is very informative for developing this model in the future.

By the metrics gained from the training phase,

a)



b)

Figure 5. Mean losses of the epochs during training (a). Mean scores of the discriminator during training (b) show how the discriminator easily overperforms the generator.

one could judge the generator to be a bit overpowered by the generator as the losses differ quite much. Also, the real and fake scores of the discriminator are getting slowly further away from each other which indicates the end of improvement and convergence failure (Fig. 3 & Fig. 5). Model didn't suffer from collapses which can be seen as a good thing.

5.1. How to increase the model's performance

Results gotten by this project indicate clear possibilities of improving the performance of the model in question. First adjustment should be done to increase generators performance. PyTorch's *nn.Sequential* is complicated to work around while reshaping layers and implementing more complicated architectures so those adjustments were spared to the later projects.

One alternative for enhancing DCGAN's performance could be inducing noise to both real and fake images during the training as distributions of fake images are usually different from the real images [14]. Also including dropout could be considered to impair the performance of the discriminator.

GAN related research is still quite novel field and in an experimental phase as new methods and architectures are frequently introduced to the scientific community. It affects the whole area of deep learning too as it is one of the most vastly developing areas of knowledge these days. The possibilities with DCGANs and other state-of-the-art GANs such as StyleGAN3 [15] are huge and should be taken advantage of in the future.

References

[1] Goodfellow Ian J., 2014, Generative Adversarial Networks, arXiv:1406.2661v1

[2] Chu Casey, et al., 2020, Smoothness and Stability in GANs, ICLR, arXiv:2002.04185v1

[3] Aggarwal Alankrita, et al., 2021, Generative adversarial network: An overview of theory and applications, *International Journal of Information Management Data Insights,* Volume 1, Issue 1, April 2021, 100004

[4] Radford, A, et al., 2015, Unsupervised Representation Learning with Deep Convolutional Generative Adversial Networks

[5] Lunarwhite, 2018, Anime Face Dataset NTU-MLDS, https://www.kaggle.com/lunarwhite/anime-face-dataset-ntumlds

[6] Van Rossum, G. & Drake, F.L., 2009. *Python 3 Reference Manual*, Scotts Valley, CA: CreateSpace.

[7] TIOBE Index for December 2021, https://www.tiobe.com/tiobe-index/

[8] Paszke, A. et al., 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., pp.8024–8035.

[9] Martín Abadi, et al. TensorFlow: Largescale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[10] O'Connor Ryan, 2021, PyTorch vs TensorFlow in 2022, AssemblyAI, https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2022/

[11] Diederik P. Kingma, Jimmy Ba, 2015, Adam: A Method for Stochastic Optimization

[12] Liu Bingqi, et al., 2021, Application of an Improved DCGAN for Image Generation, doi: 10.21203/rs.3.rs-266104/v1

[13] Tripathi Sandhya, et al., 2020, GANs for learning from very high class conditional noisy labels, arXiv:2010.09577v1

[14] Sønderby Casper Kaae, 2016, nstance Noise: A trick for stabilising GAN training https://www.inference.vc/instance-noise-a-trick-for-stabilising-gan-training/
[15] Karras Tero, Aittala Miikka, Laine Samuli, Härkönen Erik, Aliass-Free Generative Adversial Networks ,2021, *Proc. NeurIPS*