GAN network used for generation of anime faces

Ján Fabian
Faculty ITEE, University of Oulu
90014 University of Oulu, FINLAND
jan.fabian@student.tuke.sk

Abstract

Deep learning model called GAN (Generative Adversarial Network) is used to synthetize anime characters faces. GAN consist of two neural networks Discriminator and Generator which are trained together in adversarial manner. Discriminator is learning to distinguish real faces from synthetized ones, while Generator is learning how to generate more realistic faces, that are trying to fool the Discriminator. Dataset with thousands of images of anime faces is fed to the model. GAN networks have variety of potential applications from generating original anime characters, to predicting new frames in video or in the research of facial recognition systems.

1. Introduction

Generative Adversarial Network (GAN) is a type of deep learning model designed to generate new, previously unseen examples from a given dataset. It is composed of two main components: a generator network and a discriminator network. The generator network generates new examples, while the discriminator network evaluates the authenticity of the examples. The two networks are trained in an adversarial manner, where the generator tries to generate examples that can fool the discriminator, while the discriminator tries to correctly identify which examples are real and which are fake. GANs are nowadays used for a variety of tasks, such as image synthesis, text generation, and audio generation.

This paper describes the processes of creating Generative Adversarial Network capable to generate original image. The GAN was trained on dataset with images of anime character faces. Layers of Neural Networks hyperparameters were tuned. And the results analyzed.

2. Dataset

The dataset used for this project was The Anime Face Dataset NTU-MLDS. It was created at National Taiwan University as a part of course “MACHINE LEARNING AND HAVING IT DEEP AND STRUCTURED 2018 SPRING”. [1] Images were originally scrapped from www.getchu.com and various similar datasets exist. Dataset created by Brian Chao [2] consist of 63632 high quality faces. But it includes outliers such as non-human faces and badly cropped samples. Also, the shape of the images varies. However, the code for scrapping the images is included on his GitHub, so we can get better intel, how the dataset was generated.

Figure 1: Random image samples from dataset

In this work, the dataset consisting of 36 740 unique images picturing faces of anime characters was used. It was created by LUNARWHITE.[2] Characters have various face expressions and are in various poses. Pictures are in high quality and with white background. The shape

is consistent with 64x64 pixels. This makes them easy usable for various image analysis and research applications.

3. Method

In this part the process the details of the implementation process are explained. Firstly, the data have been loaded and preprocessed. Then the GAN itself was implemented, with Descriptor and Generator networks.

3.1. Preprocessing and loading of the data

Because of the relatively small size of dataset, it could be stored on the local file system, and there was no need for database queries. Custom class *AnimeData* is responsible for data handling. It implements Pytorch's Dataset class and overrides the *__init__*, *__len__* and *__getitem__* methods.

The *__init__* method sets the class properties, as *root* which is path to the folder with dataset. Property *transform* refers to Pytorch's transforms module. Property *images* holds the naturally sorted list of image sample names within the root folder.The *__len__* method returns the length of the dataset, based on the length of the *images* property.The *__getitem__* method loads individual images based on the *root* path and *index* of the image. Because image names are also integer numbers ranging from 1 to number of samples, index can be translated to the image name.

Images were preprocessed with the Pytorch's *transforms* module. They were resized by the transforms *Resize()* method to the fixed size of 64x64 pixels. Then the center of the images were cropped by the *CenterCrop()* method. Data were then transformed to Pytorch tensor with *ToTensor() transforms* method. Images were finally normalized with the *Normalize()* method with predefined *stats*.

For loading the data the Pytorch's *DataLoader* class was used. Batch size was set to 512 samples per batch. So the dataset could be split to 72 batches. The data shuffling was allowed.

3.2. Discriminator

In the training process, the Discriminator is trying to become better at distinguishing real and fake images of anime faces.

The Discriminator class is a PyTorch module that creates a discriminator neural network for use in a generative adversarial network (GAN). The class inherits from the PyTorch nn.Module class and overrides its *__init__()* and *forward()* methods.

In the *__init__()* method, the class takes in a single input, inchannels, which represents the depth of the first convolutional layer. The method uses this input to create a sequence of layers using PyTorch's *nn.Sequential* container. The sequence contains five convolutional layers, each of which is followed by a batch normalization layer and a leaky ReLU activation function. The negative slope is set to the value 0.2 according to the best practices in GAN tuning. [4] It avoids the problem of dying neurons. The final convolutional layer has only one output channel, which represents the probability whether the input image is real or fake. It is followed by a flattening layer and a sigmoid activation function.

In the forward method, the class takes in an input tensor, representing the generated image, and applies the sequence of layers defined in the init method to it. The output of this sequence is returned as the final output of the forward method, which is the label whether the image is real or not.

3.3. Generator

The Generator class is a neural network module that is used to generate images. The class inherits from *nn.Module* and has an initializer method that takes in an argument *latent_size* which is the length of the input latent vector. The class has a *nn_seq* attribute, which is an instance of *nn.Sequential*, that contains several layers of convolutional transpose, batch normalization, and rectified linear unit (ReLU) activation function. The first layer is a 2D convolutional transpose layer with input channels equal to latent_size and output channels equal to 512, followed by a batch normalization layer and a ReLU activation function. This is followed by several more layers of similar architecture, each with decreasing numbers of output channels, until the final layer, which is a 2D convolutional transpose layer with 3 output channels and a *Tanh* activation function.

The forward method takes in an input x and applies the layers of *nn_seq* to it before returning the output as a 3x64x64 Tensor image.

Not trained Generator, with random *latent* input creates only noisy RGB images.

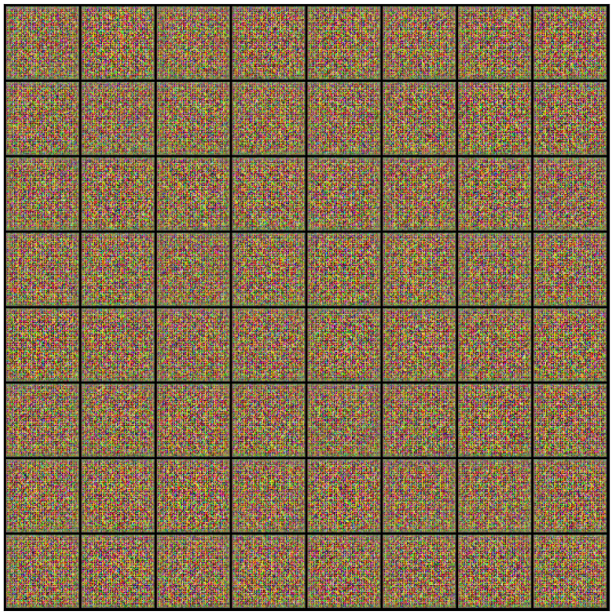

Figure 2 Generator output in initial state

During the training, the generator gets better at transforming the random *latent* input to the images of anime faces.

3.4. Loss functions and optimizers

After we have defined the networks architecture and prepared our data, to finalize the GAN model, we need to define the loss function and optimizer for both the discriminator and generator.

For the training loss calculation, two functions were defined. *Real_loss()* and *Fake_loss()*, which are used to calculate the loss of the discriminator in a GAN model.

The *Real_loss()* function calculates how close the discriminator's output is to being real by using the binary cross-entropy loss between the predicted probability of the input being real and the actual label. Binary cross-entropy loss is a common choice for the loss function of the discriminator.

The *Fake_loss()* function calculates how close the discriminator's output is to being fake by using the binary cross-entropy loss between the predicted probability of the input being fake and the actual label.

Both functions take two inputs: "preds" which is the output of the discriminator and "targets" which is the label of the input(real or fake). The function then uses the PyTorch function F.binary_cross_entropy to calculate the loss and returns the loss value.

The Adam optimizer from Pytorch's module *optim* was choice for both Discriminator and Generator networks. Learning rate was set to 0.0002 and momentum (*betas* parrameter)was set to 0.5 instead of default 0.9 as it's a common choice for GAN networks optimization.

3.5. Training

For the purpose of training, the function "*train*" was defined. It trains the GAN model made up of a discriminator (D) and a generator (G) using the given optimizers (d_optimizer, g_optimizer) for a specified number of epochs. The training process consists of two main steps: training the discriminator and training the generator.

First, the discriminator is trained on real images and fake images. Real images are passed through the discriminator, and the discriminator's output is compared to a target label of 1 (indicating that the image is real) using the binary cross-entropy loss function. Fake images are generated by the generator, passed through the discriminator, and compared to a target label of 0 (indicating that the image is fake) using the binary cross-entropy loss function.The gradients are then computed and the optimizer updates the discriminator's weights.

Next, the generator is trained to produce synthetic images that are similar to real images. Fake images are generated by the generator and passed through the discriminator. The generator's goal is to produce images that will be classified as real by the discriminator, so the label for these images is set to 1. The generator's output is compared to this target label using the binary cross-entropy loss function, and the gradients are computed and used to update the generator's weights.

For each epoch, the function also logs the loss and score for the generator and discriminator, and saves generated images and the models' weights. The code also uses the tqdm library to display a progress bar during the training process.

4. Results

During the process of training, temporal results and outputs of generator were logged. In the following sections the results of training process are analyzed.

4.1. Training losses

The model was trained on the Anime face dataset in 50 epochs. This means that the generator and descriminator went through the dataset for 50 times. This is 10 epochs more, than it was required by the task for this paper (40 epochs). The reason was to see the model behavior, when the training time of the model is extended. Selection of the larger number of epochs was limited by the computational resources of the hardware. Therefore, the model was not pushed to its limits. The behavior of Generator and Descriptor training losses stayed relatively consistent.

Figure 3: Training losses curves of Generator (blue) and Discriminator (orange) networks

As both Discriminator and Generator networks are becoming better in doing their tasks simultaneously, the loss curves are bumpy, but they follow a "horizontal" trend. The training loss of Generator oscillates along value 6 and the training loss of the Discriminator is usually less than 1. This is because when Generator learns how to generate more realistic images, the Discriminator also knows better how to distinguish them from fake ones. It is not presented in this notebook, but if we increased the number of epochs to 100 we might see a big change in the curve trends. As the Discriminator is no longer capable to distinguish between real and fake images, the Generator loss drops close to zero and Discriminator loss jumps close to 100.

4.2. Generated images

In the following figures 4-6 the improving performance of the Generator network is presented via generated images output.

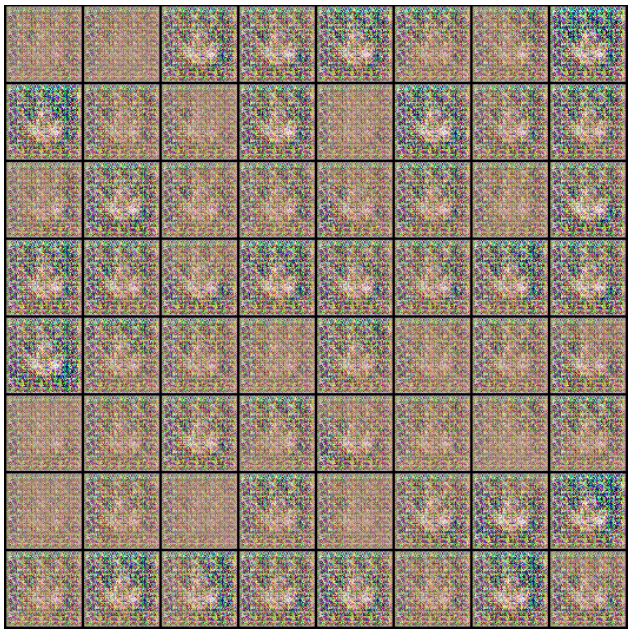

Figure 4: Generated images after Epoch 1

Even after first epoch the improvement over Figure 2 is already possible to be seen. Generated pictures are still noisy, but the dim shadow of the face is possible to see on multiple images.

Figure 5: Generated images after Epoch 25

After 25 epochs, there are for sure anime characters on the pictures, although they are no way near the original samples. Most of them lack the nose and mouth and also details in hair and eyes. Most pictures have some sort of grid at the edges.

Figure 6: Generated images after Epoch 50

After 50 Epochs, at the end of our training period, the characters are much more realistic than after epoch 25. They have more details in hair and eyes. The pictures are more consistent, without unnatural features like the mentioned grid, but they lack the clarity and details of the original samples. Small features like mouth and nose are still missing on the most of the generated samples.

5. Conclusion

In conclusion, a GAN network was trained on a dataset of anime character faces in order to generate new, original images. The dataset used for this project was The Anime Face Dataset NTU-MLDS, which consisted of 36,740 unique images of anime character faces. The GAN network was composed of a generator and a discriminator network, which were trained in an adversarial manner. The generator network was tasked with generating new images that could fool the discriminator network, while the discriminator network was tasked with identifying which images were real and which were fake. Through fine-tuning of the network's hyperparameters and preprocessing of the data, the GAN was able to generate realistic and diverse images of anime characters.

Despite good results, the anime faces can still be more realistic and diverse. One way to improve this model would be to increase the model size of the generator and the discriminator. Larger models have more capacity to learn the complexities and nuances of the data and produce more realistic and diverse samples. Experimenting with different architectures, such as using deeper or more convolutional layers, can also lead to better results. Fine-tuning the model on a specific task or using a larger and more diverse dataset can also lead to improvement in the generated samples. Additionally, using different types of loss functions and changing the hyperparameters can also lead to improvement. These changes may also require more epochs to make us of the maximum performance of the model.

References

[1] "MACHINE LEARNING AND HAVING IT DEEP AND STRUCTURED 2018 SPRING" *https://speech.ee.ntu.edu.tw/~hylee/mlds/2018-spring.php*

[2] Brian Chao, Anime Face Dataset, *https://github.com/bchao1/Anime-Face-Dataset*

[3] LUNARWHITE, *Anime Face Dataset NTU-MLDS, https://www.kaggle.com/datasets/lunarwhite/anime-face-dataset-ntumlds*

[4] Manish, Nayak, Medium, Nov 22, 2018, *https://medium.datadriveninvestor.com/deep-convolutional-generative-adversarial-networks-dcgans-3176238b5a3d*