Report on-
Flower Images Generator using GAN

Md. Masudul Haque- 1620370042
&
Al Shahriar-1621011042

Department of Electrical & Computer Engineering

North South University

Date: 13/05/2020

CSE465 | Pattern Recognition

Instructor's Name: Mohammad Ashrafuzzaman Khan

Abstract

Generative Adversarial Networks (GANs) have shown impressive results of generating synthetic images. In current computer science world GAN is the hot topic. As part of CSE465 project, we studied how to use various GAN models to generate flower images. In particular we developed DCGAN model which can generate fake image of flowers on the basis of input real flower images. Potential future improvements could make our system more generic with expand usable data sources.

Our project available at GitHub

Keywords: GAN, DCGAN, PyTorch, Jypyter Notebook, Fake flower Image.

Table of Contents

List of Figures

List of Tables

1. Introduction

In 2014, Ian Goodfellow introduced the Generative Adversarial Network (GAN) as a novel technique to generate samples from a target probability distribution from which a data set is already available. The GAN works by an adversarial process, in which two units called the Generator and the Discriminator. Where the generator creating samples of "fake" data and the discriminator distinguishing between real and fake data. At each step, the generator updates its samples in a way that it may "fool" the discriminator into classifying them as genuine. After a sufficient number of steps, if the data set provided is large enough and the architectures in both units are the suitable, the samples generated by the generator start resembling the real data, and improve with each step of the game.
The DCGAN (Deep Convolutional GAN) developed by Alec Radford, Luke Metz and Soumith Chintala (2016) has been used extensively for the generation of images. In this model, each of the generator and the discriminator is a deep convolution neural network (CNN). Several architectures, additional features, and modifications have been incorporated into the DCGAN, and it has been found to perform well on most popular image data sets.
We trained a Deep Convolutional generative adversarial network (DCGAN) to generate new Flower images after showing it pictures of many real Flower images. For this model we use PyTorch which is an open-source machine learning library for Python. We run our model in Jupyter Notebook

2. Background & Related Work

In "DCGAN--Image Generation "paper [1], they explored the potential of deep learning to generating real like images. They used Deep Convolutional Generative Adversarial Network (DCGAN) which has proven to be a great success in generating images. They have discussed the theoretical aspect of GAN and also discussed about their methodology to create a DCGAN Model for MNIST Datasets and CelebA Datasets.
Qiaojing Yan & Wei Wang [2] used deep convolutional generative adversarial networks (DCGAN) to do various image processing tasks such as super-resolution, denoising. DCGAN allows them to use a single architecture to do different image processing tasks and achieve competitive PSNR scores. While the results of DCGAN shows slightly lower PSNR compared to traditional methods, images produced by DCGAN is more appealing when viewed by human. DCGAN can learn from big datasets and automatically add high-frequency details and features to images while traditional methods can't. The generator discriminator architecture in DCGAN pushes it to generate more realistic and appealing images.

3. Dataset

Finding an appropriate dataset is one of the most important tasks for this work. We used two datasets in our project. We collected data set from Flower Datasets. There were two category of datasets, one was "17 category dataset" and other one was "102 category dataset". We combined them into one single dataset. The final dataset consists of over 10K total images. Some preprocessing was needed. All the images were converted to .jpg format.

4. Hardware Requirement & Software Installation

In this section we are going to describe hardware requirement & software installation process.

4.1. Hardware Requirement

Our workstation Computer system are consisting of AMD Ryzen 7 2700X CPU, Nvidia RTX 2060 GPU, 16 GB RAM and SSD which helps us to run our program smoothly.

4.2. Software Installation

As previously said, we use PyTorch open-source learning library for Python and we run our project on Jupyter Notebook. There are many versions of PyTorch but some of these are not working for our project. The PyTorch versions that have worked for us are:

PyTorch Build:	Stable (1.5)
OS:	Windows
Package:	Conda
Language:	Python
CUDA:	10.1

Table 1- Software Version

Installing the Jupyter Software was not a hard task. JupyterLab can be installed using "`conda install -c conda-forge jupyterlab`" in windows command prompt.

5. Methodology & Algorithms

We divided the whole process into several steps which are elaborate bellow:

5.1. Inputs

We used many inputs. Some define inputs are:

`dataroot`	The path to the root of the dataset folder.
`workers`	The number of worker threads for loading the data with the DataLoader
`batch_size`	The batch size used in training. The DCGAN paper uses a batch size of 128
`image_size`	The spatial size of the images used for training. This implementation defaults to 64x64.
`nc`	Number of color channels in the input images. For color images this is 3
`nz`	Length of latent vector
`ngf`	Relates to the depth of feature maps carried through the generator
`ndf`	Sets the depth of feature maps propagated through the discriminator
`num_epochs`	Number of training epochs to run. Training for longer will probably lead to better results.
`lr`	Learning rate for training. As described in the DCGAN paper, this number should be 0.0002
`beta1`	Beta1 hyperparameter for Adam optimizers. As described in paper, this number should be 0.5
`ngpu`	Number of GPUs available. If this is 0, code will run in CPU mode. If this number is greater than 0 it will run on that number of GPUs

Table 2- Inputs

5.2. Data

We create a directory named *Flower* and extract the zip file into that directory. Then, set the *dataroot* input for this notebook to the *Flower* directory. The resulting directory structure should be: "C:\Users\masudulhaque\Flower". This is an important step because we will be using the ImageFolder dataset class, which requires there to be subdirectories in the dataset's root folder. Now, we create the dataset, create the dataloader, set the device to run on, and finally visualize some of the training data.

Figure 1-Training Data Samples

5.3. Implementation

With our input parameters set and the dataset prepared, we can now get into the implementation. We will start with the weight initialization strategy, then talk about the generator, discriminator, loss functions, and training loop in detail.

5.3.1. Weight Initialization

From the DCGAN paper, the authors specify that all model weights shall be randomly initialized from a Normal distribution with mean=0, stdev=0.2. The weights_init function takes an initialized model as input and reinitializes all convolutional, convolutional-transpose, and batch normalization layers to meet these criteria. This function is applied to the models immediately after initialization.

5.3.2. Generator

The generator, G, is designed to map the latent space vector (z) to data-space. Since our data are images, converting z to data-space means ultimately creating an RGB image with the same size as the training images (i.e. 3x64x64). In practice, this is accomplished through a series of strided two-dimensional convolutional transpose layers, each paired with a 2d batch norm layer and a relu activation. The output of the generator is fed through a tanh function to return it to the input data range of [-1,1]. It is worth

noting the existence of the batch norm functions after the conv-transpose layers, as this is a critical contribution of the DCGAN paper. These layers help with the flow of gradients during training. An image of the generator from the DCGAN paper is shown below.

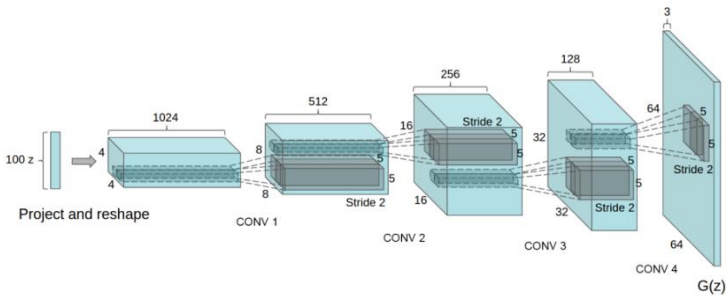

Figure 2-DCGAN Generator

Notice, the how the inputs we set in the input section (*nz*, *ngf*, and *nc*) influence the generator architecture in code. *nz* is the length of the z input vector, *ngf* relates to the size of the feature maps that are propagated through the generator, and *nc* is the number of channels in the output image (set to 3 for RGB images).

5.3.3. Discriminator

As mentioned, the discriminator, D, is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real (as opposed to fake). Here, D takes a 3x64x64 input image, processes it through a series of Conv2d, BatchNorm2d, and LeakyReLU layers, and outputs the final probability through a Sigmoid activation function. This architecture can be extended with more layers if necessary, for the problem, but there is significance to the use of the strided convolution, BatchNorm, and LeakyReLUs. The DCGAN paper mentions it is a good practice to use strided convolution rather than pooling to downsample because it lets the network learn its own pooling function. Also, batch norm and leaky relu functions promote healthy gradient flow which is critical for the learning process of both G and D.

5.3.4. Loss functions & 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 (BCE Loss) function which is defined in PyTorch as:

$$\ell(\mathrm{x},\mathrm{y}) = \mathrm{L} = \{\mathrm{l1},\dots,\mathrm{lN}\}\top, \mathrm{ln} = -[\mathrm{yn}\cdot\log\mathrm{xn} + (1-\mathrm{yn})\cdot\log(1-\mathrm{xn})$$

Notice how this function provides the calculation of both log components in the objective function (i.e. log(D(x)) log(D(x)) and log(1−D(G(z))) log(1−D(G(z)))). We can specify what part of the BCE equation to use with the y input. This is accomplished in the training loop which is coming up soon, but it is important to understand how we can choose which component we wish to calculate just by changing y (i.e. GT labels).

Next, we define our real label as 1 and the fake label as 0. These labels will be used when calculating the losses of D and G, and this is also the convention used in the original GAN paper. Finally, we set up two separate optimizers, one for D and one for G. As specified in the DCGAN paper, both are Adam optimizers with learning rate 0.0002 and Beta1 = 0.5. For keeping track of the generator's learning progression, we will generate a fixed batch of latent vectors that are drawn from a Gaussian distribution (i.e. fixed_noise). In the training loop, we will periodically input this fixed_noise into GG, and over the iterations we will see images form out of the noise.

5.3.5. Training

Finally, now that we have all of the parts of the GAN framework defined, we can train it. Be mindful that training GANs is somewhat of an art form, as incorrect hyperparameter settings lead to mode collapse with little explanation of what went wrong. Here, we will closely follow Algorithm 1 from Goodfellow's paper, while abiding by some of the best practices shown in ganhacks. Namely, we will "construct different mini-batches for real and fake" images, and also adjust G's objective function to maximize $logD(G(z))$ log(G(z)). Training is split up into two main parts. Part 1 updates the Discriminator and Part 2 updates the Generator.

5.3.5.1. Training the discriminator

Recall, the goal of training the discriminator is to maximize the probability of correctly classifying a given input as real or fake. In terms of Goodfellow, we wish to "update the discriminator by ascending its stochastic gradient". Practically, we want to maximize-

$$log(D(x)) + log(1-D(G(z))) \; log(D(x)) + log(1-D(G(z))).$$

Due to the separate mini-batch suggestion from ganhacks, we will calculate this in two steps. First, we will construct a batch of real samples from the training set, forward pass through D, calculate the loss ($log(D(x))$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 ($log(1-D(G(z))$ log(1−D(G(z)))), and accumulate the gradients with a backward pass. Now, with the gradients accumulated from both the all-real and all-fake batches, we call a step of the Discriminator's optimizer.

5.3.5.1. Training the Generator

As stated in the original paper, we want to train the Generator by minimizing $log(1-D(G(z)))$ log(1−D(G(z))) in an effort to generate better fakes. As mentioned, this was shown by Goodfellow to not provide sufficient gradients, especially early in the learning process. As a fix, we instead wish to maximize $log(D(G(z)))$ log(D(G(z))). In the code we accomplish this by: classifying the Generator output from Part 1 with the Discriminator, computing G's loss *using real labels as GT*, computing G's gradients in a backward pass, and finally updating G's parameters with an optimizer step. It may seem counter-intuitive to use the real labels as GT labels for the loss function, but this allows us to use the $log(x)$log(x) part of the BCELoss (rather than the $log(1-x)$log(1−x) part) which is exactly what we want.

Finally, we will do some statistic reporting and at the end of each epoch we will push our fixed_noise batch through the generator to visually track the progress of G's training. The training statistics reported are:

- ✓ **Loss_D** - discriminator loss calculated as the sum of losses for the all real and all fake batches *($log(D(x))+log(D(G(z)))$log(D(x))+log(D(G(z))))*.
- ✓ **Loss_G** - generator loss calculated as $log(D(G(z)))$ log(D(G(z)))
- ✓ **D(x)** - the average output (across the batch) of the discriminator for the all real batch. This should start close to 1 then theoretically converge to 0.5 when G gets better. Think about why this is.
- ✓ **D(G(z))** - average discriminator outputs for the all fake batch. The first number is before D is updated and the second number is after D is updated. These numbers should start near 0 and converge to 0.5 as G gets better.

6. Results and discussion

Finally, let's check out how we did. First, we will see how D and G's losses changed during training. Then look at a batch of real data next to a batch of fake data from G.

6.1. After 5 Epochs:

Below is a plot of D & G's losses versus training iterations after 5 epochs-

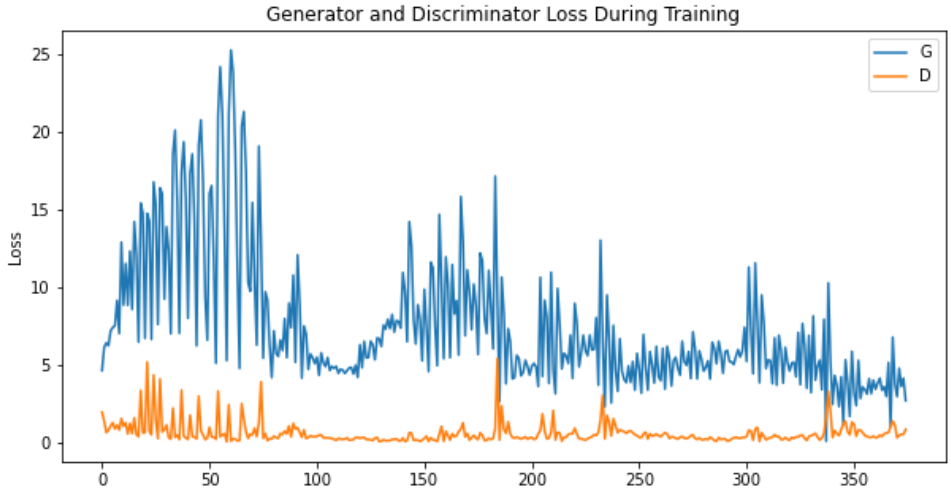

Figure 3-Plot of D & G's losses versus training for 5 epochs

Real images and fake images side by side after 5 epochs-

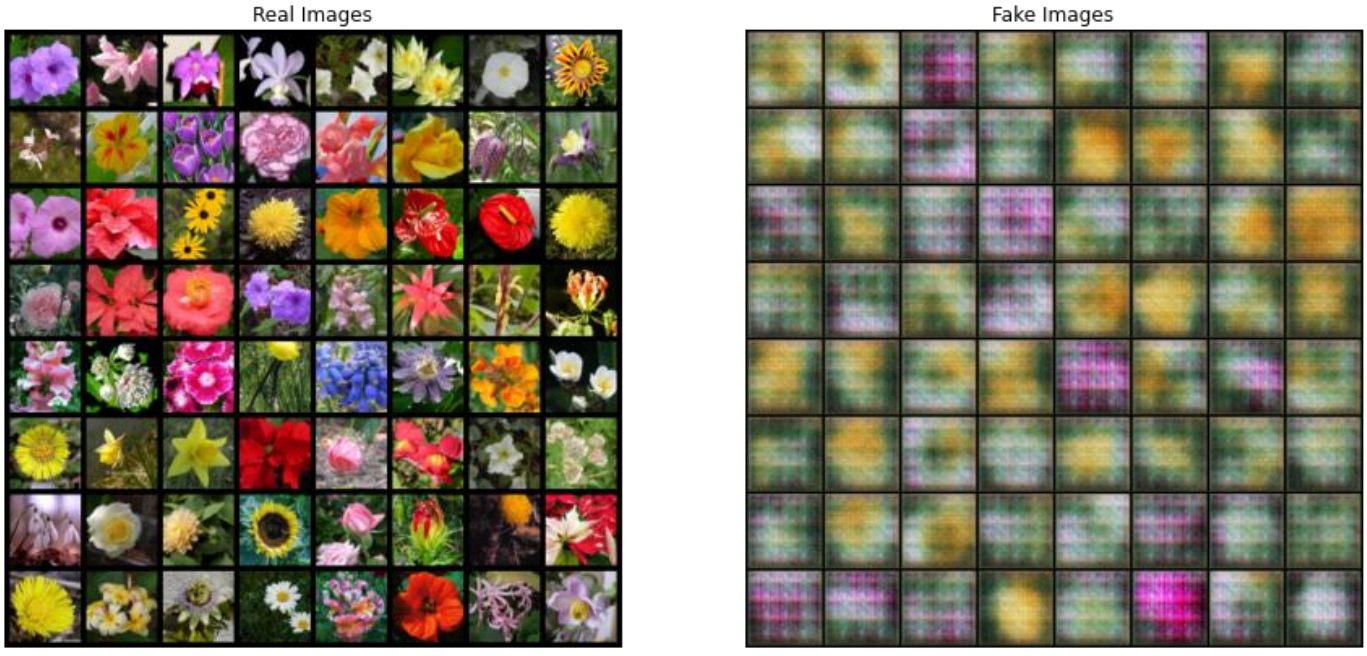

Figure 4-Real images and fake images side by side for 5 epochs

6.2. After 50 Epochs:

Below is a plot of D & G's losses versus training iterations after 50 epochs-

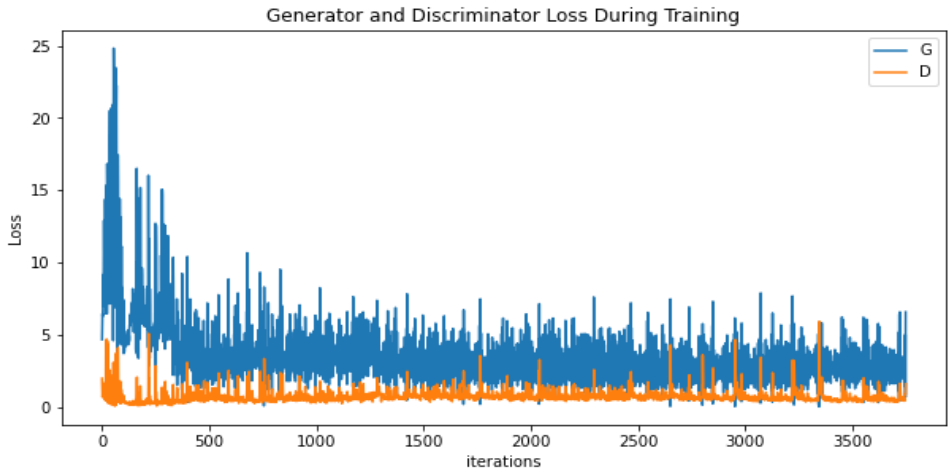

Figure 5-Plot of D & G's losses versus training for 50 epochs

Real images and fake images side by side after 50 epochs-

Figure 6-Real images and fake images side by side for 50 epochs

6.3. After Final 500 Epochs:

Below is the final plot of D & G's losses versus training iterations after 500 epochs.

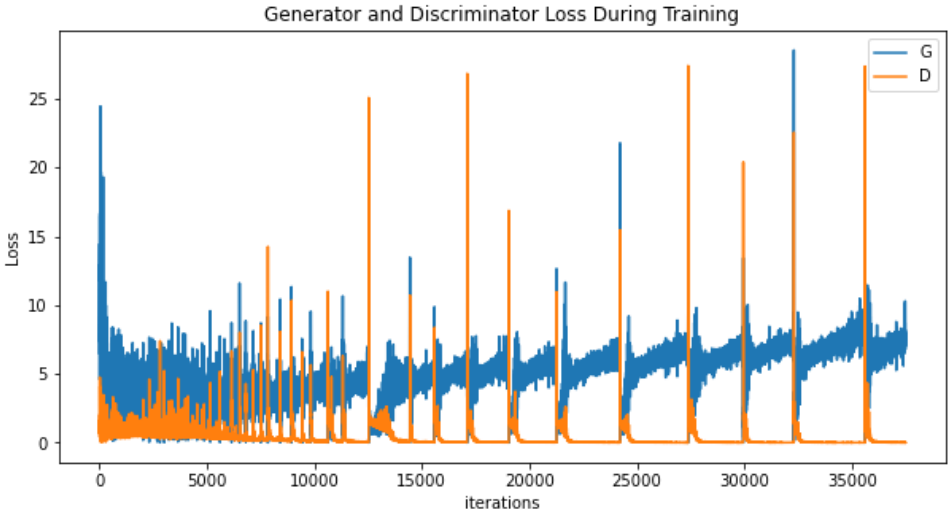

Figure 7-Plot of D & G's losses versus training for 500 epochs

Finally, let's take a look at some fake images after ***500 epochs***.

Figure 8-Final Fake Images

From this YouTube link we can see the Visualization of G's progression of our project!

7. Future Work

We have reached the end of our journey, but there are several places we could go from here like- we can train for longer to see how good the results get, modify this model to take a different dataset and possibly change the size of the images and the model architecture. We can generate any kind of fake images on the basis of input image. After few modifications this model can be used in medical images diagnosis and so on.

8. Conclusions and recommendations

We proposed a DCGAN Flower images generating model which can be used in many sections like Artificial Intelligent, Medical analysis etc. This model also can provide huge amount of data for future machine learning projects. But DCGAN can give good results on single faces dataset (flowers, animals, human faces etc.) but it is not suitable to use for more complex dataset such as natural scenes. For this we have to move towards more complex models

9. Acknowledgements

This section allows authors to acknowledge contributors and other sources that are not appropriate to list in the references section. Example:

This work was conducted under Grant No. 12345, administered by X. The authors are also particularly grateful to Dr. Jane Smith for her insight into the nature of Y.

10. References

[1] Chapagain, Ashutosh. (2019). DCGAN--Image Generation. 10.13140/RG.2.2.23087.79523.

[2] Yan, Q., & Wang, W. (2017). DCGANs for image super-resolution, denoising and debluring. *Advances in Neural Information Processing Systems*, 487-495.

Appendix A: Place the title of appendix here

Provide appropriate appendices as necessary. Each appendix should begin on a new page.