SpongeGAN Squarepants: A Generative Spongebob Model

Michael Fielder
University of Central Florida
College of Engineering and Computer Science
Department of Computer Science
MichaelFielder@knights.ucf.edu

Abstract

With the rise in popularity of deep learning methods, came with it in an increase in the performance of procedurally generated methods. Indeed, everything from noise to images have seen major improvements in the way that they are able to synthesize samples. This is mainly due to the GAN architecture coined in the early 2010's which used a generator-discriminator architecture in order to generate synthetic images. The objective of this project is to leverage the DCGAN architecture on a set of frames from the television show Spongebob Squarepants in order to see if it is able to generated convincing unique images. From my experimentation, I am able to show that the model is able to generate overfitted yet convincing samples even with the limited training time available.

Google Colab Link: https://bit.ly/38yInEC

1. Introduction and Background

1.1. Problem Statement

The formal problem statement for this program is to create a generative network that is trained on frames from the Spongebob Squarepants television show and is able to generate unique image in the output. While I go into more detail into how GANs work later in this section, I can say here that this project can be categorized as a *classification* problem.

1.2. Problem Importance

While this problem isn't important in the sense that it will cure a disease or expedite the creation of self-driving cars, it is still important because creativity is an integral part to the human condition. Entertainment in the form of paintings or television shows are a way to provide enjoyment to a person's life. The problem with any art is that it require skill and time to create. If we are able to generate art without the need for an expert, then we can create more art to entertain at a faster rate.

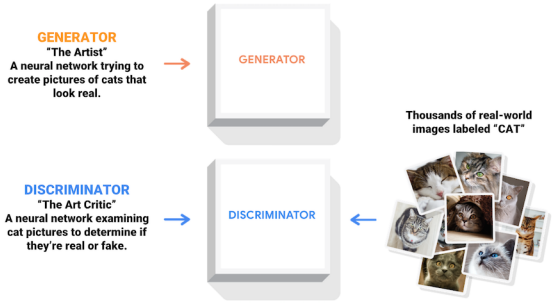

Figure 1. An illustration of the GAN Art Analogy [3]

2. Previous Work

2.1. GAN Networks and DCGAN

In June of 2014, Ian Goodfellow and company proposed their seminal architecture known as a Generative Adversarial Network [1]. The objective of a GAN is to simultaneously train two models at the same time: a generative model G and a discriminative model D. The idea is that G attempts to emulate the data distribution of the dataset whilst D attempts to discriminate between real samples from the data and "fake samples" generated by G. A common analogy used for explaining the GAN model is that of the art forger and the art critic. AS shown in Figure 1 the generator G acts as a forger who attempts to make a forgery that could fool a seasoned art critic. When he starts, the forger is inexperienced at his craft and creates fakes that would never fool the critic. However, overtime the forger gets better at creating fakes to the point where the critic is unable to distinguish between a real picture and a fake one.

For a GAN to succeed, it should have an effective generator that is able to distinguish between a real and fake, otherwise the generator will not be able to create convincing samples. In fact, the main reason that this architecture is

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}\big[log D(x)\big] + \mathbb{E}_{z \sim p_z(z)}\big[log(1 - D(G(z)))\big]$$

Figure 2. Value function of adversarial nets [1]

able to work is that discriminators have become more effective in more recent times allowing for better generators to be developed. Ultimately, a GAN is said to succeed if it is able to make the binary classification probabilities of both classes 0.5. This means that the discriminator G is unable to distinguish between a real and fake image, and is essentially guessing if it is real of fake.

The next big advancement in GAN models was the creation of the Deep Convolution GAN (DCGAN) [2]. This models biggest development was the in the paper Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks. It can be seen as a direct continuation of the GAN architecture with the biggest difference being that it uses convolutional layers in its execution. Convolutional layers have shown improvement in a lot of different deep learning applications including here in generative models.

2.2. How Adversarial Nets Work

It is important to delve into the main objective of an adversarial net in general. According to Goodfellow's original GAN paper [1]. In general, the paper explains that D and G are playing a "min-max" game with each other. D is attempting to *maximize* the probability of guessing a correct label (fake or real) while G is attempting to minimize the probability that D guesses it's a fake. This is expressed in Figure 2 which shows the value function.

3. Dataset

3.1. Overview

The dataset I use is the *Cartoon Classification Dataset* from Kaggle by the user Volkan Özdemir [4]. This dataset contains thousands of examples of animation frames from various shows such as Adventure Time, Family Guy, and the target of this project, Spongebob Squarepants. Since I only needed examples from Spongebob, I download and compiled together every frame present in both the Training and Testing set of the dataset. Figure 3 shows some examples of frames in the dataset

3.2. Pre-Processing

In order for the data to be effective in the model, the images needed to be pre-processed in order to be handled by the model effectively. The main issue is that the frames provided were stored in a 4:3 resolution. This means that black bars are present in the images. This poses a problem, because I didn't want the DCGAN to learn them during training. My solution to this was to crop the images horizontally

Figure 3. A 4x4 grid sample of the dataset

Figure 4. An example of the black bars in the original dataset

to remove them. By doing this, I am able to only include the important information of the image while utilizing a method that doesn't distort it. An example of the black bars can be seen in Figure 4.

Another thing that I did to the dataset was institute a image numbering convention for their filenames. An images filename indicates their number in the data and is stored using leading zeros (Ex. 0000 and 0123). This naming was mostly done for ease of directory navigation and general organization.

3.3. Technical Details

In total, there are exactly 12970 images in the dataset stored in the `jpeg` image format. Each image has a size that is close to 300 x 300 pixels. While the model reduces the size of the image in training and generation, I opted to keep a larger size in case it is used in another application.

4. Model and Code

The general model used in my paper is a DCGAN architecture that is able to take in RGB images of size $(128 \times 128 \times 3)$ and generate an image of the same size.

4.1. DCGAN Codebase

For my project, I specifically use an edited version of the DCGAN using Nathan Inkawhich's *DCGAN Tutorial* found on *PyTorch's* website [5]. The original application of this network was to generate fake celebrity headshots using a DCGAN network. I decided to use PyTorch instead of TensorFlow, because I am more accustomed to its workflow since it tends to be used more in academia.

4.2. Loss Function and Optimizer

For my model, I used `nn.BCELoss()` which PyTorch's version of Binary Cross Entropy Loss. This is used because there are only two classes present in training, real and fake images. For the optimizer, I used the Adam optimizer, which in PyTorch is `optim.Adam()`.

4.3. Alterations

The first alteration that had to be made to the original model is to change the the generator and discriminator so that it is able to handle images of size 128×128 instead the original 64×64. After some experimentation, the simplest way to do this was to include another `nn.ConvTranspose2d` into its generation. The same was also done for the discriminator. This is because each Convolutional Transpose outputs to a size of $A/2 \times 2B \times 2B$ if the size of the input to the layer was $A \times B \times B$. Since each block in the generator also included a `nn.BatchNorm2d` and a `nn.ReLU` layer, I decided to also include those after the additional transpose layer.

The other change I did to the model was to include a means of saving the the G and D model parameters and the loss lists G and D. This was done so that I could save the model after training since it took multiple sessions. For saving the lists, I used the `pickle` library and for the model parameters, I used PyTorch's `torch.save()` and `torch.load()` functions on each model's parameters.

4.4. Training Loop

In my project, the model trains on separate batches of exclusively either real or fake images. This is to make sure that the training of the discriminator is simplified. The outputs of the batch are calculated using the BCE Loss and the model is updated using the Adam optimizer. Another thing to note is that the discriminator is trained before the generator, because it is important for the discriminator to be well-trained so that the generator can create good fake images to trick the discriminator.

5. Experimentation

5.1. Overview

My model was trained on a batch size of 16 for 15 epochs. 3 - 4 training session needed to be performed with each session doing 5 epochs. The Adam optimizer used a learning rate of 0.0002 although a rate of 0.002 was also tried but caused the loss the rise too quickly. My model was trained on Google Colab Plus using a V100 GPU. Total training time was approx. 30 hours where each training session was run overnight. The exact numbers are difficult to measure since I implemented the save feature after running some test runs.

5.2. Training

Figure 5 shows how the model is training overtime. In the early epochs, the model has no understanding of what a model looks likes and therefore its outputs are close to random noise. As the model trains more, its output is more distinct and representative of a scene in Spongebob. Granted, even the later epochs still possess some noise due to the nature of adversarial networks.

5.3. Loss Results

The loss for both the D and G both lowered during training and stabilized. G received a minimum loss of 0.874 and D 0.00197. The loss overtime for a GAN model is expected to fluctuate overtime and settle at some specific value. Figure 6 shows the losses of G and D over training. Another thing to note is that D's loss is lower than G's which is also expected during training.

5.4. Testing

Testing was simple for this. Since there is no objective empirical to measure the results of a GAN, I loaded the most recent checkpoint and passed in random noise to the generator G and observed the results. Figure 7 shows the output of G. As shown in the output, the generator is able to produce convincing frames albeit with some added noise. The way to reduce this noise would be to train the model on more epoch so that it learns a better representation of the data.

6. Analysis of Results

One of the biggest difficulties of GAN architectures is the tendency of model to overfit on data if the data isn't sufficient for the task. In my model, I believe that the model is too overfitted to the dataset, causing less interesting results. There are a number of different reasons for this phenomenon including an unfocused dataset, a lack of samples, and the embedding size. First I will address the over-variety of frames. In other applications of GANs, the dataset is more consistent. For example, some will constrain the input to only include headshots of inputs. In my dataset, there

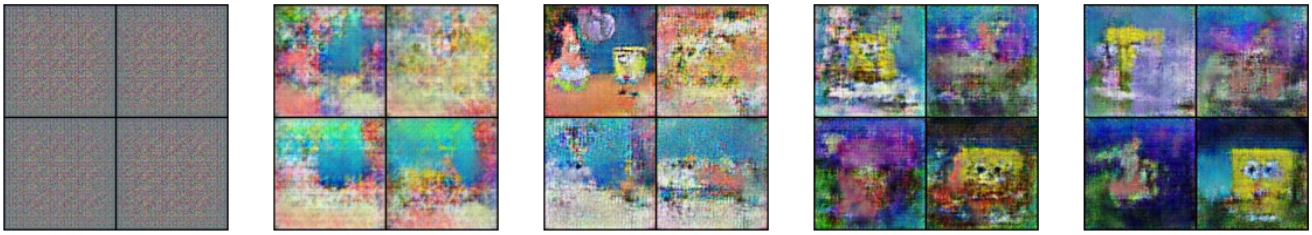

Figure 5. Example of generated images in different epochs

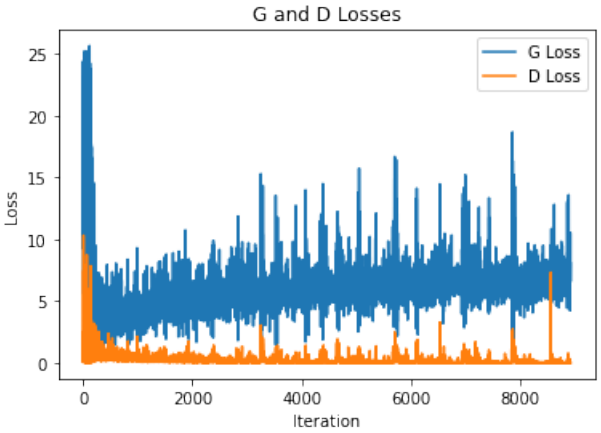

Figure 6. Graph showing the G and D loss over training

Figure 7. A 4×4 sample output of the generator

are images with multiple characters, close-ups, objects, ect. The next issue is that it includes frames that are very similar. I suspect this is because the creator of the original dataset simply download adjacent frames in a scene. Since there are usually 24 frames a second in a TV show, this can result

Figure 8. An example of similar frames

in frames that are similar in appearance. This can be seen in Figure 8 where each is a separate entry into the dataset. The final problem is the embedding size which is 64 in this application. Since I used a larger image size, keeping the same embedding size could result in less variety of output. With less variety of output comes overfitting.

7. Potential Future Work and Solutions

The first thing I would address in my application would be do clean up the samples in my program. Since there were 12k images in the database, it's very hard to clean up the database with the time allotted. In a perfect world, it would be best to have a more focused databased as well. My solution to this would be to create a facial recognition model that can identify faces in the show and use this to create cropped samples. If I were to sample more frames from the show, I would most likely do this by downloading digital versions of the episode and randomly sample the frames rather than sequentially sample them in order to increase the variety of the dataset and improve training.

8. Conclusion

Overall, I believe that my project was a success. I was successfully able to train a DCGAN on frames from Spongebob. I would say that the model produced specifiable output given the time-frame and computational constraints. With all this being said, there are still things that could be improve in the pipeline, the biggest issue being the distribution of the dataset. With all this in mind. I would say that I learned a lot through this project and am grateful for the opportunity to learn more about adversarial networks.

9. References

1. https://arxiv.org/abs/1406.2661
2. https://arxiv.org/abs/1511.06434
3. ttps://www.tensorflow.org/tutorials/generative/dcgan
4. https://www.kaggle.com/datasets/volkandl/cartoon-classification
5. https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html