Training Generative Adversarial Networks on limited data

Luka Sabotič

Abstract

For this project I tackled a problem of training GANs on a small dataset. I used multiple different stochastic augmentations to try and prevent the discriminator overfitting, while not leaking augmentations into the generator. I used StyleGAN2 architectures for my models and trained them on the Oxford flowers-102 dataset. I couldn't train the models for long enough to get good results, but pattern shows that the approach might work if training would be run for longer.

1 Introduction

Generative Adversarial Networks (GANs) have shown remarkable success in synthesizing realistic images across a wide range of domains. However, training GANs typically requires large-scale datasets to achieve high-quality outputs and stable convergence. In many practical scenarios such as medical imaging, art, or scientific domains, collecting and annotating large datasets is expensive, time-consuming, or even infeasible. Especially for a student without any appropriate hardware doing all training on Google Colab like me.

In this project, I explore techniques for training GANs under limited data conditions, following the approach presented in Training Generative Adversarial Networks with Limited Data by Karras et al. (2020) (1). The paper proposes several key innovations aimed at improving GAN performance in data-constrained settings, including adaptive discriminator augmentation (ADA), which dynamically adjusts data augmentations based on the discriminator's confidence, helping prevent overfitting without reducing diversity in generated samples.

My goal is to implement and evaluate the ADA-based GAN training pipeline, understand the challenges of learning from only a few examples and evaluate the visual quality and stability of generated images.

2 The problem

Training generative adversarial networks to obtain an image generator requires a very large training set. This is usually not a problem, as there are a lot of very large image datasets available online and transfer learning also enables us to use a pretrained model and fine-tune it on a smaller dataset. But sometimes we want a generator to only produce images from a certain domain without the influence of images from other domains that might have been included in the pretraining datasets. And if the target domain does not possess a sufficiently large and available dataset, we are in trouble. If we use a too small training set, the discriminator will overfit and no matter how good the generator's fakes are, the discriminator will always know they are fake, because it "remembered" the whole training (real) set.

One might immediately think of augmentations, but as soon as we add augmented images to the training set, this will impact the generator's output. So we need to approach this carefully – we need to sufficiently confuse the discriminator, whilst training the generator to only take into account the images in the training set.

3 Adaptive discriminative augmentations

The trick for not allowing the discriminator to learn which augmentations we are performing is to add probability. By using $p < 1$ as the probability of making each augmentation, the discriminator cannot know whether it was performed or not and cannot help itself if it finds the augmentation's features – maybe they are just present in the real image. The second randomization used is the magnitude of the augmentations. For each augmentation, there is a distribution selected for its parameters. They are then sampled randomly to determine the strength of the augmentation applied. This makes it significantly harder for the discriminator to recognize the augmentation and even if it does, it can't really tell whether it is an augmentation or the real image.

The authors in the original paper spent quite some time experimenting with the augmentation probability. This p is used for all augmentations independently, so it makes sense for it to be a small value. They found that it is best to use an adaptive value instead of a constant one for the entire training. To calculate p updates, we need to set a few hyperparameters: a sliding window size, the discriminator's target confidence, a heuristic overfitting measure and over how many images we want the p to travel from 0 to 1, named ada_num_imgs. The idea is to update p after processing the sliding window size images and change its value by a constant Δp, depending on

the discriminator's output. Δp is calculated by

$$\Delta p = \frac{\text{sliding window size}}{\text{ada\_num\_imgs}}$$

. As the heuristic overfitting measure, I chose the $r = \mathbb{E}(D)$ over the sliding window, precisely the average number of correctly classified real images. I chose the goal for this value to be 0.6. The authors provide another option for the heuristic overfitting measure, but I chose this one, as it is less sensitive to other hyperparameters, and I didn't have the time to search for the optimal combinations. Here, I have to admit I misread the instructions in the paper and am calculating the update of p after every batch, with a slightly larger sliding window, but this mistake only means I am doing more computations, the effect of the method should be the same. In the end, the sign of Δp is determined by $(\text{int}(r > 0.6) \cdot 2 - 1$, so positive if the discriminator is overconfident and negative if it is not confident enough.

All together, there are 18 transformations, split into two categories. Geometric and color transformations, visualized in figure 1 and image-space filtering and corruptions, visualized in figure 2. Every transformation is applied with probability p, which is the same for all the transformations for the given batch. The magnitudes of the transformations are sampled for each one separately.

I mentioned that simply enlarging the training set with augmentations is going to leak the augmentations into the generator and the generated images are going to show signs of applied transformations. This is why in this approach, I applied augmentations to both the real image and the generated fake. In the paper, authors mention that it is not clear why this approach works, while other, similar approaches fail or leak augmentations, but do not explain why that is. My thinking is that this is due to the discriminator being shown not a distribution of training images, but the augmented distribution of them. Because the augmentations are not invertible (they are stochastic), discriminator cannot figure out the real distribution. The generator, on the other hand, tries to replicate the distribution of the training images, but because its outputs are also augmented in the same way, it is unknowingly replicating the original, training distribution. The trick is that we can transform the generator's distribution, because we are able to calculate the augmentations, but the discriminator cannot figure out the original distribution, because the inverse of the augmentations does not exists.

4 Methods

The original paper already comes with code, but it is written for experimental purposes and using the TensorFlow library. I wanted to implement the pipeline myself and being more used to PyTorch, I started from scratch. I still followed the guidelines of the authors and chose StyleGAN2 (2) as my GAN architecture. The authors recommended this as it appeared more stable during training of this type compared to other GAN variations. The training is simple: pick and augment a batch of images form training set, generate and augment a batch of fake images, calculate the generator and the discriminator losses and perform a step on both optimizers. I used XX images in the training set, Adam optimizers and a learning rate of XX. For the training set I used Oxfords 102 flowers dataset (3). It contains images of 102 flower classes usually found in the United Kingdom.

4.1 StyleGAN2

I implemented 3 models. The most simple one is a mapping network, which takes a randomly sampled vector from $\mathcal{N}(0, I)$ and puts it through 8 fully connected layers, producing w. This is then used as an input to the generator.

The generator takes w as an input and per-channel noise for later injection. Then it goes layer by layer, starting with a constant tensor and upsampling, doing style demodulation and a toRGB operation. This is repeated as many times as needed to reach the desired resolution, in my case 128x128. Style demodulation consists of two steps. First, it uses w from the mapping vector, does a learned affine transformation on it to produce a style vector. This is then used to modulate convolution kernels of the generator. Then, the kernels get normalized or demodulated, to prevent spikes in their sizes – keep weight size variance low and avoid unusual artifacts in the images. ToRGB layers are $1x1$ convolutions that map the current latent image into 3 channel RGB image. RGB layers also include style modulations. Additionally to the generator's loss, a path length penalty is used. This is a loss that ensures nice interpolation of images by checking whether a small change of W in the latent space also results in a small change of the generated image: $\frac{\partial g(x)}{\partial w}$ needs to be small. This is a computationally intensive computation, so it is only done every 16 steps.

The discriminator is a simple CNN decoder with skip connections. Its architecture is a reverse of the discriminator's.

5 Results

As mentioned above, I was running my training on Google Colab's free GPUs. I was interchanging 3 separate Google accounts to train my models and lost count of how many epochs I managed to do, but it was a little over 100. The training set size I used was 8189, so I generated a little over 818.900 images with the generator. Its final version is available on the linked GitHub repository.

The results are the images like the ones in figure 3. They are not very good, but it is clear that the images are getting close to flowers, so I am optimistic. The colors and color ratios are correct and most of the time, textures are correct as well. Upon a closer inspection, one can see that there is a sufficient level of detail in the image, shapes are not monotone and large, but tiny patches of texture appear, which could become tiny details such as the pistil or grass in the background. With some imagination, one can see the shapes of flowers forming bellow the pixelated cover, but as I observed the training procedures, results are still drastically improving, suggesting that I might get good results with more training. I unfortunately could not afford that. Keep in mind that the images appear lower-quality, because I used a resolution of 128 pixels, to reduce training time.

6 Conclusion

Regardless of the failure to get good results, I am optimistic this approach works really well. The sequence of generations of all epochs nicely show that the generator is improving. The authors of the original paper got very good results, albeit on a higher resolution, so I am convinced that I would too if I ran the training for longer. Click here for the GitHub repository with project code.

References

[1] KARRAS, T., AITTALA, M., HELLSTEN, J., LAINE, S., LEHTINEN, J., AND AILA, T. Training generative adversarial networks with limited data, 2020.

[2] KARRAS, T., LAINE, S., AITTALA, M., HELLSTEN, J., LEHTINEN, J., AND AILA, T. Analyzing and improving the image quality of stylegan, 2020.

[3] VISUAL GEOMETRY GROUP, U. O. O. The oxford flowers 102 dataset. https://www.robots.ox.ac.uk/~vgg/data/flowers/102/, 2025. Accessed: 2025-06-22.

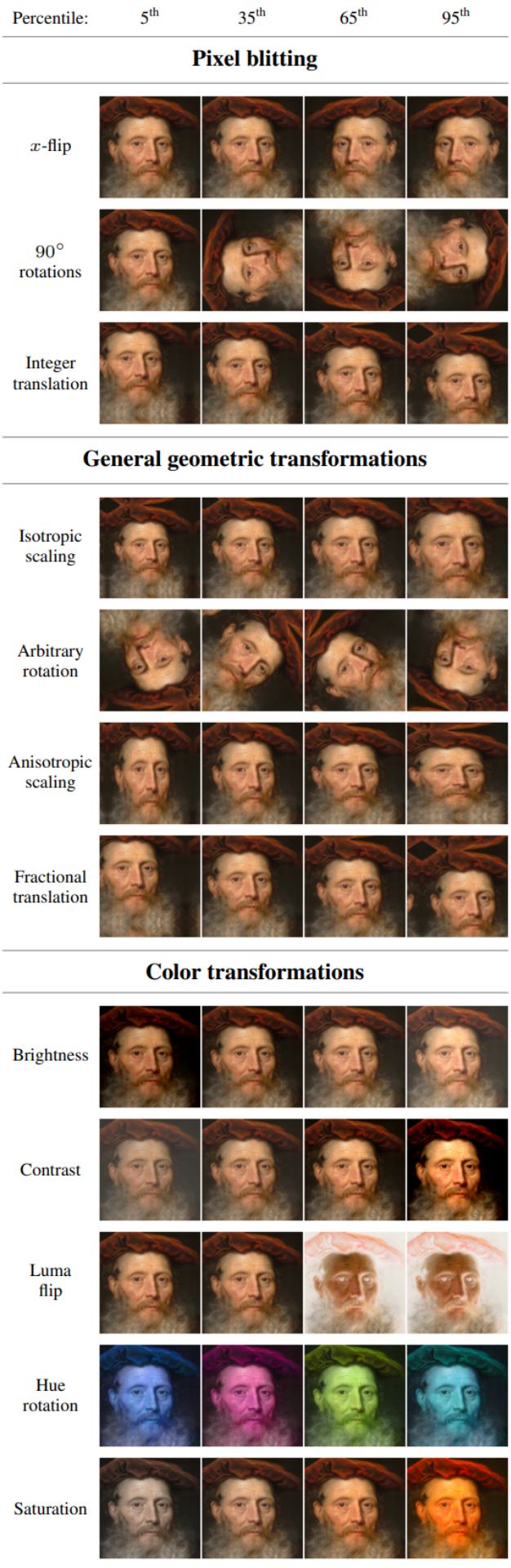

Figure 1: Effects of geometric and color transformations

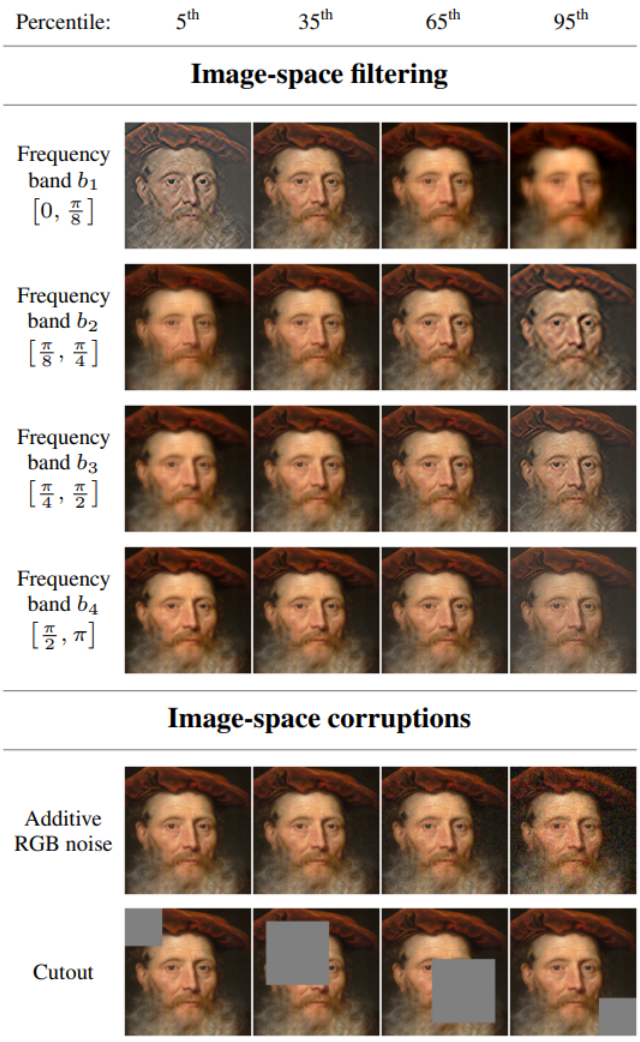

Figure 2: Effects of image-space filtering and corruptions

Figure 3: Images generated with a generator trained for a little over 100 epochs