CS63 Fall 2019
Generating Doodle Faces using GANs

JJ Balisanyuka-Smith and Francesco Massari

12/12/2019

1 Introduction

Generative Adversarial Networks, in short GANs, were invented by Ian Goodfellow in 2014 [3]. In the context of the symbolic Minimax algorithm, for instance, the idea of using competition in AI has been around for a long time. GANs are a fairly recent application of game theory to subsymbolic AI. The central idea is that two players are trying to maximize or minimize a certain value, depending on the perpective. GAN's work by pitting two deep neural networks, a discriminator and generator, against each other. The discriminator is trained to predict whether an image is an actual instance of, for example, a face doodle, while the generator is trained to take in random input and turn it into an image that is likely to be classified as legitimate. The two networks are trained alternatingly taking typically thousands of training steps.

The recent interest in GANs is partly due to their ability to produce stunnig images such as the scarily convincing StyleGan2 [4]. Moreover, they are often used to make "deep fakes", fake images of people or objects that are difficult to tell from real images. Part of the process of training a generator is creating a network that can discriminate between real and fake images that a even human eye cannot tell apart. Ideally, the products of successful GAN training are (1) a very robust discriminator and (2) a network that generates realistic images from noise. Some count GANs as an example of, at least partially, unsupervised learning. Even though they need large amounts of positive examples, a key component of GAN-training is the process of the two networks learning from each other.

For our project, we chose to generate 28×28 pixel grayscale pictures of a doodle face, inspired by more computationally expensive projects that have succeeded to generate pictures of human faces. We hope to give a simplified proof of principle of the idea that GANs can learn to reproduce various types of complex patterns.

2 Methods

2.1 Data and Preprocessing

We used data from Google "Quick, Draw!", an online database of user generated doodles that Google uses to train AI [5]. Images in "Quick, Draw!" can be conveniently downloaded as a numpy bitmaps [5]. Each image is a one-dimensional numpy array of length 28×28 containing grayscale values that range from 0 to 255 and to normalize all values by dividing them by 255 [5].

Since our aim was to generate faces, we downloaded the `"full_numpy_bitmap_face.npy"` dataset and used the first 100000 images of face doodles (Fig. 1) to train our network [5]. The only preprocessing necessary to run our network on the data was to reshape the set into the multidimensional form $(10000, 28, 28, 1)$.

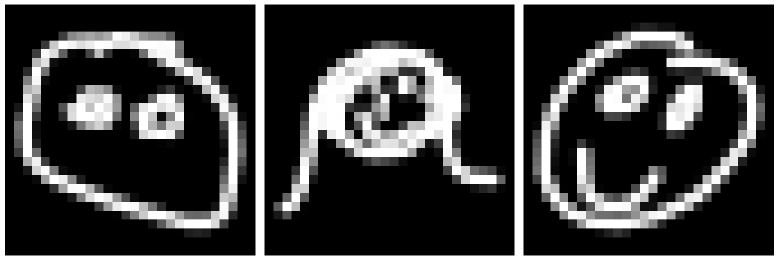

Figure 1: examples from the dataset

2.2 Final Architecture

Since GAN-training is extremely sensitive to hyperparameters, many sources recommend starting with a model that has already proven successful on a similar task. Our GAN was built using code by Rowel Atienza [1], which was originally designed to generate images of digits in the style of MNIST, and taylor it to our data. Following the rationale outlined in the introduction, our GAN consists of two networks, a classifier and a generator, the architecture of which is shown on the table below [1].

layer type	number of features	kernel size	stride	activation-function
Conv2D	64	5	2×2	LeakyReLU ($\alpha = 0.2$)
Conv2D	128	5	2×2	LeakyReLU ($\alpha = 0.2$)
Conv2D	256	5	2×2	LeakyReLU ($\alpha = 0.2$)
Conv2D	512	5	1×1	LeakyReLU ($\alpha = 0.2$)
Dense (1 unit)	-	-	-	sigmoid

Table 1: Discriminator architecture.

As shown in Table 1, our discriminator consists of four two-dimensional convolutional layers to which we added dropout of 0.6 [1]. The dropout forces our model to generalize and prevents it from merely relying on a small set of features, as for example the presence of eyes. Padding was set to "same" in all layers to avoid loss of information at the borders of the image [1]. The activation function used for all convolutional layers was LeakyReLU, which has an extended range compared to the classic rectified linear unit, so that there is a gradient even when the activation is negative. Thus, Leaky ReLU helped us avoid the problem of a vanishing gradient. The value of the negative slope coefficient α was adapted from the original code, since our goal was to start with hyperparameters that had previously proven successful [1]. The optimizer used for training is RMSprop with a learning rate of 0.0002 and a decay of 6^{-8}, whereas the loss function is "binary crossentropy" [1].

All training hyperparameters were adapted from the original code, since subsequent experiments did not motivate substantial changes [1].

layer type	filters	kernel size	activation-function
Dense	-	-	relu
Conv2DTranspose	32	5	relu
Conv2DTranspose	16	5	relu
Conv2DTranspose	8	5	relu
Conv2DTranspose	1	5	sigmoid

Table 2: Generator architecture.

The generator contains a dense layer and three two-dimensional transpose convolutions, all of which are followed by batch normalization with a momentum of 0.9 [1]. The central feature of transpose convolution is that it increases the dimensionality of the input, instead of decreasing it. In our case we move from an array of 100 random values to a grayscale image of size 28×28. Batch normalization has the advantage of stabilizing and speeding up training, while avoiding over-fitting [2]. High momentum is recommended for small batch sizes, like ours [6]. As in our discriminator model, we used padding so that the entirety of the generated image would be equally informative [1]. The dense layer is followed by an instance of Reshaping to ensure matching dimensions and dropout of 0.4 for reasons analogous to the ones described in the last section. The first convolutional layer is preceeded by an instance of two-dimensional Upsampling, which doubles the dimensionality of the input [1]. Finally, the output layer consists of a one-feature convolution which outputs the desired grayscale image of size 28×28. The generator uses the same loss-function and optimizer as the discriminator with the only difference that it has a higher learning rate (0.0002) and a slower decay (3^{-8}) [1]. This difference gives the generator the chance to keep up with the discriminator.

Our GAN is trained as follows: the generator is fed random noise to generate a series of non-face images [1]. These images are then combined with a random subset of the face doodles from Google "Quick, Draw!" and the resulting training batch is used to train the discriminator [1]. All generated images are treated as negative and all doodles as positive examples [1]. Subsequently, the adversarial model, a joint model consisting of the generator at the top and then the discriminator at the bottom, is fed new noise and is trained to tweak its weights such that it maximizes the probability of its output being classified as a face by the discriminator [1]. Hence, at each training step, the discriminator is trained on 512 images (half real, half generated) and the generator on 256 images (all noise) [1]. Generally, training should last for as long as it takes to generate images of acceptable quality, but due to time constraints, we limited our training to 10000 steps.

2.3 Preceding Attempts

Our first attempt at coding a GAN was to only adapt the rough features of code by Rowel Atienza [1] and essentially code the architecture and training from scratch using our intuitions and prior knowledge. One key difference to the final version of our code, which will be explained below, is that we pre-trained the discriminator on a mixed dataset gathered from Google "Quick, Draw!".

For this dataset, we combined 5000 instances of face doodles, with 5000 instances of doodles from 20 distinct categories. When training the full adversarial model, we combined generated images with the mixed dataset to train our discriminator. The following section will outline why we ultimately had to work more closely with Atienza's code.

3 Results

3.1 First Try (bad news first!)

The first thing our experiments revealed is how important hyperparameters are in ensuring success. Even small discrepancies between our first attempt and our source code led to substantial differences in image quality. We discovered how even very small changes to our architecture led to considerably different final outputs.

Changing the activation function in the dense layers of our discriminator to sigmoid increased validation accuracy from 50% to 94%. This is presumably, because sigmoid is better suited for categorical classification, such as deciding if a doodle is a face or not. We then changed the number of nodes in our dense layers from 50 to 128, which increased validation accuracy to 96% for 3 independent runs. The final major changes we made were to add dropout and change the optimizer from Stochastic Gradient Descent(SGD) to RMSProp. Neither of these lead to a more than a 1% change in validation accuracy, but they sped up training, so we kept both.

Once we had completed our discriminator model we tried to train our generator. We made many changes to our overall model in order to achieve acceptable results but we were ultimately incapable of finding the correct parameters. The experiments showed that the generator produced artifacts in the form of vertical lines which appeared instantly and remained throughout training. Unfortunately, we were incapable of finding the exact source of these artifacts. Finally, we realized that incremental changes to our personal hyperparameters did not substantially impact the generated outputs (Fig. 2).

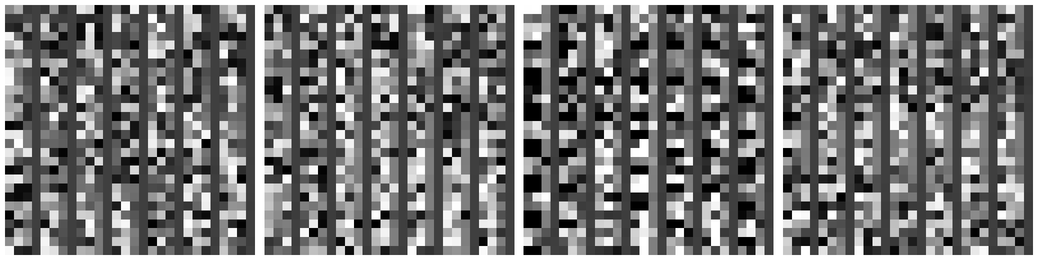

Figure 2: Virtually constant output over many runs with different hyperparameters

3.2 Second try (good news second!)

Since we were able to validate Atienza's model by running it on the MNIST dataset, we chose to adapt his code to fit our data. The results of training the new model on the MNIST data set for 10000 steps, which it had been designed for, are shown in Figure 3.

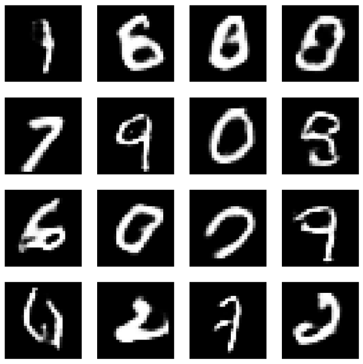

Figure 3: Output of the generator after 10000 steps of training on MNIST

Using the new architecture, we obtained visibly improved results (Fig. 4). Results of comparable quality were obtained over three independent runs, thus inspiring confidence that the performance of our final GAN did not depend to much on the weight initializations of the networks (Fig. 4).

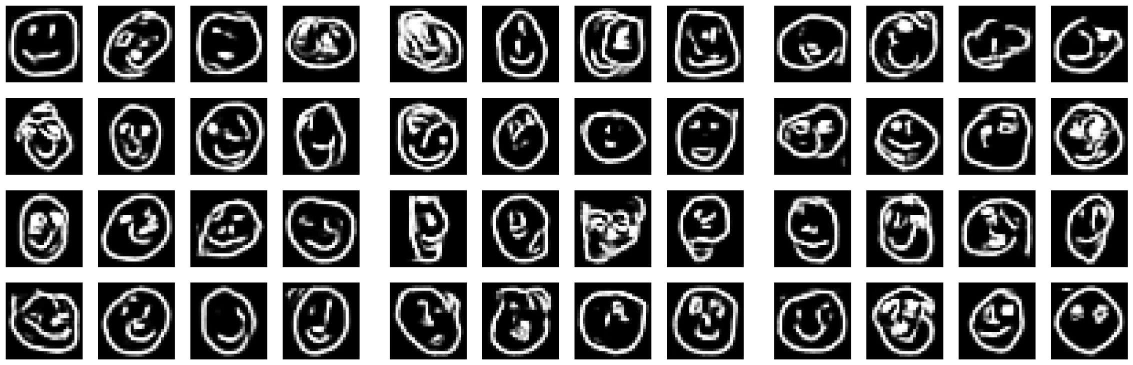

Figure 4: Generator outputs after three independent runs with 10000 training steps each

The competition between discriminator and generator is most apparent when looking at the evolving accuracy and loss of both models over the course of 10000 steps of training (Fig. 5). A glance at the graphs in figure 5 reveals that both the validation accuracy and the loss of the generator, which reflect how well it fools the discriminator, are highly unstable, whereas the validation accuracy of the discriminator seems to quickly converge to a values of around 0.5 (Fig. 5). The loss of the discriminator behaves similarly (Fig. 5). This shows that both models steadily improve up to about 2000 training steps, where they reach a point equilibrium after which further training does not lead to substantial changes (Fig. 5).

Comparing figure 5 to graphs generated from an earlier run of an unsuccessful model (Fig. 6) shows that it is not easy to assess the success of GANs quantitatively. The differences between the graphs are hard to interpret and one might even feel inclined to say that the graphs in figure 6 look better, since the they look more stable. However, visual inspection of the images generated by the GAN behind figure 6 reveals the superiority of our final model (Fig. 7).

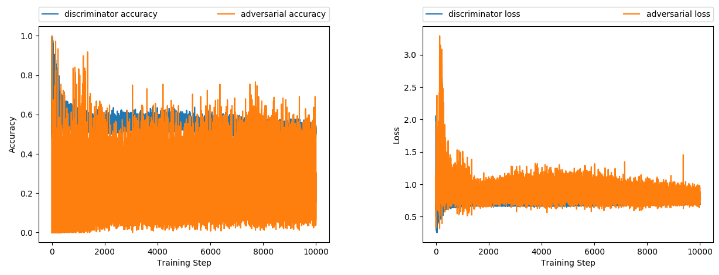

Figure 5: Accuracy (left) and loss (right) of the GAN trained on face doodles for 1000 steps

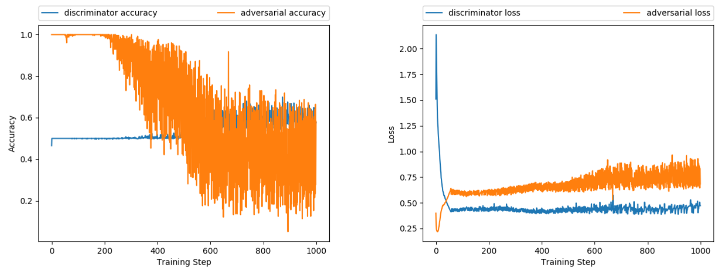

Figure 6: Accuracy (left) and loss (right) of a failed GAN trained on face doodles for 10000 steps

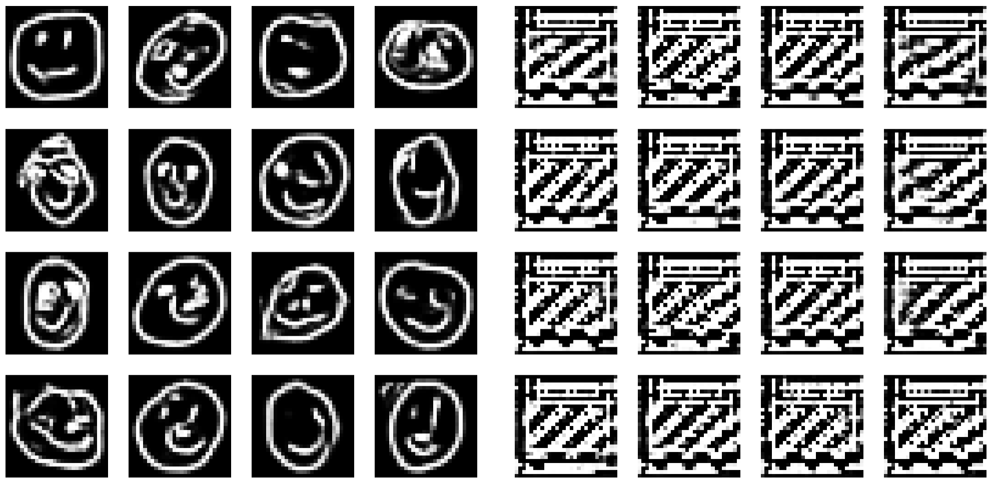

Figure 7: Generator outputs of our final model (left) and an earlier model (right) with similar quantitative success statistics.

The difficulty of quantitatively measuring the success of GANs results from the fact that loss and validation accuracy only reflect the success of one half of the GAN relative to the other half. The "ultimate classifier" a GAN has to usually defeat is the human brain, since humans are the ones to decide whether the output "looks realistic". Hence, qualitative analysis of generator output appears to be the most important way to assess GAN success. A particularly intriguing aspect of our generated images is the fact that they feature distinct "drawing styles" (Fig. 4). While most generated smileys have simple dots as eyes, one of them has has two circles (Fig. 4, second square, fourth row, fourth image). Other smileys have distinct features, such as an open mouth (Fig. 4, second square, second row, fourth image) or eyebrows (Fig. 4, second square, third row, third image). Whether this is an instance of artificial creativity is debatable, since the generated features purely rely on learnt features, but it at least leads to interesting variability in the generator output.

4 Conclusions

In summary, it deserves to be reemphasized that, as we have seen, GAN performance is highly sensitive to hyperparameters, the precise influence of which is poorly understood even by experts. Thus, we have found it an effective approach to rely on published network architectures and training procedures that can subsequently be adapted as needed.

The improvement of our GAN stagnated quickly, but yielded qualitatively clearly recognizable results. Hence, our model can be said to learn fairly quickly. Importantly, visual inspection seemed to us to be the preferable method to evaluate the GAN, since quantitative analyses can be inconclusive. Finally, the generated images featured remarkable variability and showed a rich variety of ways to represent the same objects. This last fact can be provocatively interpreted as a tangible instance of machine creativity, or at least simulated creativity.

References

[1] Atienza, Rowel. *GAN by Example using Keras on Tensorflow Backend*. Apr. 2017. URL: `https://towardsdatascience.com/gan-by-example-using-keras-on-tensorflow-backend-1a6d515a60d0`.

[2] D, F. *Batch normalization in Neural Networks*. Oct. 2017. URL: `https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c`.

[3] Goodfellow, Ian J. et al. *Generative Adversarial Networks*. 2014. arXiv: `1406.2661 [stat.ML]`.

[4] Karras, Tero, Laine, Samuli, and Aila, Timo. *A Style-Based Generator Architecture for Generative Adversarial Networks*. 2018. arXiv: `1812.04948 [cs.NE]`.

[5] *Quick, Draw! The Data*. URL: `https://quickdraw.withgoogle.com/data`.

[6] R, Ilango. *Batch Normalization - Speed up Neural Network Training*. June 2018. URL: `https://medium.com/@ilango100/batch-normalization-speed-up-neural-network-training-245e39a62f85`.