Implementation of Generative Adversarial Network (GAN) with PyTorch

Jacob Thrasher
West Virginia University
Eberly College of Arts and Sciences
Morgantown, WV
jdt0025@mix.wvu.edu

Dr. Marjorie Darrah
West Virginia University
Eberly College of Arts and Sciences
Morgantown, WV
marjorie.darrah@mail.wvu.edu

***Abstract*—Generative Adversarial Networks (GANs) are generative models specialized in creating synthetic images from randomly sampled noise. Two convolutional neural networks compete in a two player non-zero sum game in which a generator attempts to create new images to fool a discriminator that determines whether a given image is real or fake. This Pytorch implementation was trained on the CelebA dataset to generate human faces from a normally distributed noise sample.**

I. Background

A. Generative Adversarial Networks (GANs)

Initially proposed by Ian Goodfellow in 2014, GANs are generative models involving two independent neural networks that compete in a game [1]. These networks are referred to as the generator G and discriminator D. While GANs are primarily used to create and manipulate images, they have also been useful in a variety of other fields as covered in 1.5 Survey of GANs. For the purpose of this paper, we will focus on the specific task of image generation.

The aforementioned game is played as follows: The generator and discriminator are trained in tandem. G is given randomly generated noise, z, and attempts to create a batch of new images $G(z)$. A combination of these "fake" images and "real" images, x, are given to the discriminator. It then attempts to classify each image as real or fake and both networks update their weights accordingly. D is rewarded for all correct guesses and G is rewarded every time it is able to fool D. An important distinction here is that G is not rewarded if D incorrectly labels a real image. This follows basic logic as the generator did not play a role in the output of $D(x)$. Instead, G is rewarded each time $D(G(z))$ is incorrect. The pseudocode that defines the training loop can be seen in **Algorithm 1**. The game is ideally played until a Nash equilibrium is achieved.

More technically, we wish to train the generator such that the probability distribution p_G obtained by $G(z)$ approaches p_{real}, the probability distribution from $D(x)$. Simultaneously, D is trained to maximize the probability of correctly labeling samples both from the dataset and from G [1]. This is modeled as a two player minimax non-zero sum game with a value function:

$$min_G max_D V(G, D) = E_x[log(D(x))] + E_z[1 - log(D(G(z)))] \quad (1)$$

Here, D attempts to maximize the entire function $V(G, D)$, whereas G minimizes only the right operand.

$$E_z[log(1 - D(G(z)))] \quad (2)$$

The value function is simply a derivation of Binary Cross Entropy (BCE). This is clear because the value for D is simply the sum of binary functions: $BCE[D(x)] + BCE[D(G(z))]$. It follows then that G is derived from the function: $BCE[D(G(z))]$ A full derivation can be found in **Appendix A**.

It can be observed that (2) tends to perform poorly in practice. Early in training, when G is poor, D is able to reject $G(z)$ with high confidence. Because of this $D(G(z))$ will approach zero, causing the output of $log(1 - D(G(z)))$ to approach zero as well, saturating the function [1]. A solution proposed in the original GAN paper by Goodfellow et al. was that instead of having G attempt to minimize (2) it should instead maximize $log(D(G(z))$.

GANs are trained by alternating optimization on the G and D. Over training the discriminator before updating the weights on G will prevent the generator from improving as it will never fool D and thus never learn. Additionally, under training D will cause the GAN to reach a Nash equilibrium too early as the generator will quickly and easily fool D even while producing "garbage" images. The method used to alternate training is largely up to the developer. Typical approaches include either performing one iteration on the discriminator, then one on the generator or performing k iterations on D before doing one on G. Though, there is some debate on which of these methods performs better [2].

B. Problems with GANs

Training GANs can be very temperamental and hard to train. Unlike normal neural networks which are simply trained to evaluate static data, GANs are more of a balancing act. Both G and D must continuously counteract improvements made by the other, creating a very unstable environment. As a result,

Algorithm 1 G, D tandem training with k steps pretraining

for Total iterations **do**
  for Total Batches of size B **do**
    $X \leftarrow \{x_1, x_2, ..., x_B\}$
    $Z \leftarrow \{z_1, z_2, ..., z_B\}$
    $G_z \leftarrow \{G(z_1), G(z_2), ..., G(z_B)\}$
    for k steps **do** Update D by maximizing:
    $-\nabla\theta_D \frac{1}{B}\sum_{i=1}^{B}[log(D(X)) + log(1 - D(G(Z))]$
    end for
   Update G by minimizing:
    $-\nabla\theta_G \frac{1}{B}\sum_{i=1}^{B}[log(1 - D(G(Z))]$
  end for
end for

there are a variety of "failure modes" associated with GAN training.

The most obvious GAN failure is that of convergence. The performance of each network is heavily dependent on the performance of the other. A strong discriminator can prevent the generator from learning and vice versa. Even more bothersome is that as G improves, D will naturally have a more difficult time distinguishing between real and fake images, eventually approaching 50% accuracy. This means that the discriminator's feedback will become less useful over time [3]. If this scenario is reached, the discriminator will end up giving garbage feedback to the generator which could serve to deteriorate the quality of generated images in the future. As stated by [3], "for a GAN, convergence is often a fleeting, rather than stable, state".

It is also possible that the generator could produce an especially convincing image that fools the discriminator easily. Then in order to reproduce those results, it will create a very similar image again. After a few iterations of this, G may begin to produce only that image, resulting in very little diversity in its outputs. The discriminator may learn to reject the redundant images, but it also may not. In the situation where the generator produces many very similar images and the discriminator cannot accurately reject these images it is apparent that G and D have arrived at a local minimum. This failure mode is presented as the "Helvetica scenario" in the original GAN paper [1] but is more commonly referred to as mode collapse [3]. **Figure 1** demonstrates visually how mode collapse can look.

A great deal of research has been conducted seeking a solution to the problem of mode collapse. One notable solution is minibatch standard deviation [6]. This algorithm attempts to solve the Helvetica scenario by allowing D to compare all images across the entire minibatch, which helps ensure adequate variety is maintained throughout training. This particular algorithm is advantageous over other solutions as it introduces no new trainable parameters or hyperparameters.

First, the standard deviation for each feature in each spatial location is computed across the entire batch. Let w, h, l represent width, height, and layers respectively. Given a batch

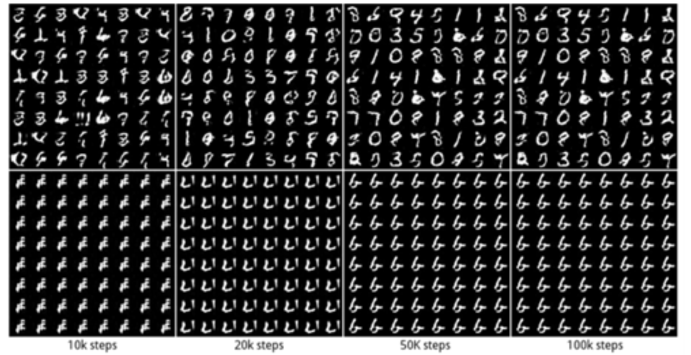

Fig. 1. Example of a mode collapse on MNIST dataset.

of size m comprised of feature maps $f \in \mathbb{R}^3$, the standard deviation of each $f_{i,j}$ across the entire batch is computed. This operation yields $w * h$ standard deviations, which are all averaged to a single value, std_{avg}. Finally, std_{avg} is replicated and concatenated to all spatial locations across the entire batch. The Python implementation of this algorithm can be found in **Appendix B**. Mathematically, these values can be computed as follows:

$$mean_{i,j} = \frac{1}{m}\sum_{l=1}^{m} f_{i,j,l} \quad (3)$$

$$std_{i,j} = \sqrt{\frac{1}{m}\sum_{l=1}^{m}(f_{i,j,l} - mean_{i,j})^2} \quad (4)$$

C. Performance metrics

Evaluating the performance of GANs differs significantly from traditional methods. The problem with generating new and unique images from some randomly sampled noise is that there is no "label" for which to compare the output. The task at hand is not to create an image of some specific instance, but instead to create an image that resembles the training dataset in some way. As such, an algorithm is needed to determine the overall quality of a generated image.

There are two primary performance metrics: Inception Score (IS), and Fréchet Inception Distance (FID). Functionally, these two approaches are very similar, but the methods with which they evaluate performance differ subtly.

Inception Score (IS) evaluates two things simultaneously: variety across multiple images, and individual images look like something [7]. It is based on the Inception classifier built by Google. Inception returns a probability distribution. This distribution can then be used to determine if the image contains a distinguishable object. For example, if there is a clear subject, the probability distribution will be skew toward a certain category, otherwise it will be relatively uniform. It should be noted that we do not care if the actual category is correct. We simply want Inception to detect a clear object.

By repeating this process for some 5000 images (as recommended by the original authors) and summing the label

distributions we can create a marginal distribution. Ideally, similar labels should have a skewed distribution, indicating that the subjects are all very similar. Conversely different labels should have a more uniform distribution, which indicates variety across images **Figure 2** shows a visual representation of the behavior of IS.

In an ideal situation, the aforementioned distributions should be opposites, so we obtain our Inception Score by computing the Kullback-Leibler (KL) Divergence of the two probability distributions. A high KL divergence indicates very different distributions, thus giving desirable results. Mathematically, Inception Score can be computed via the following function:

$$IS = \exp KL[p(y|x = G(z))||p(y)] \tag{5}$$

where:
x = set of images
y = set of labels
$p(y|x = G(z))$ = label distribution from InceptionV3
KL = Kullback Leibler Divergence defined by:

$$D_{KL}(P||Q) = -\sum_{x \in X} P(x) log(\frac{Q(x)}{P(x)})$$

Like Inception Score, Fréchet Inception Distance uses the Google Inception architecture as a performance measurement tool. Unlike IS, FID measures the structural similarity between two instances. Instead of using the probability distribution generated by Inception, FID uses the output from the layer before prediction to compare the feature maps across an image batch. This is done by comparing the mean and standard deviation of each of the feature maps using Fréchet Distance [5]. Real and fake images are passed into the Inception network, and the resulting embeddings, R and F respectively, are given to the below function, computing the FID score for the given image set.

$$d^2 = (\|m_R - m_F\|_2)^2 + Tr(C_R + C_F + 2(C_R C_F)^1/2) \tag{6}$$

where:
m_R = feature-wise mean of real images
m_F = feature-wise mean of fake images
C_R = covariance matrix of real images
C_F = covariance matrix of fake images

A desirable model would produce a low FID score as it implies the distance between real and fake images is as small as possible.

II. Methodology

This implementation of GAN is trained on the CelebA dataset to generate human faces. CelebA is a database that contains over 200,000 images of 10,000 celebrities in various poses. It also contains attribute annotations such as eyeglasses, smiling, bangs, etc. As we are only focused on generating generic faces we can ignore the annotations. It should be noted that some implementations of GANs such as Conditional GANs (CGANs) can generate images with specific attributes based on certain specifications. Futher discussion on this topic can be found in **IV. Future Works**.

A. Loading data

The process of loading the image data into memory is simple. The images provide by CelebA are of size 178×218 pixels. Since GANs become more unstable as images become larger, all elements are first resized to 64×64 pixels. The data are then converted to a PyTorch tensor object. Finally, the images are normalized such that all pixels fall into the range $[-1, 1]$. Typically, pixel values would be normalized to $[0, 1]$, but since G uses tanh as its output, we must account for possible negative values.

Images are stored as 3D matrices, where each pixel is represented by a vector containing red, green, and blue values. Color values are represented as a number in the range $[0, 255]$, where 255 is the maximum intensity. For example, the color red can be represented as the vector $< 255, 0, 0 >$. To normalize, we begin by center the RGB values such that they fall into the range $[-127, 127]$, then divide everything by 127.

B. Model Architecture

Both G and D are fully convolutional neural networks, which means no linear layers are used for output. G accepts input noise of size $[batch_size, latent, x, y]$, where $batch_size$ corresponds to the number of images to be synthesized and $latent$ represents the depth, or number of input channels. The trailing x and y represent the horizontal and vertical dimensions of the initial input. During experimentation, these values were set to 128, 100, 1, and 1 respectively. This means 128 images are generated simultaneously based on an a single one dimensional vector of length 100. The hidden layers in G are comprised of transposed convolutional layers. Unlike typical strided convolutions, which decrease output dimension, transposed convolutional layers increase the output dimensions according to the following function:

$$D_{out} = (D_{in} - 1) * S - 2 * P + (K - 1) + 1 \tag{7}$$

where:
D = image dimension
S = stride size
P = padding amount
K = kernel size

Each hidden layer was initialized with the following values: $S = 2, P = 1, K = 4$. Additionally, the number of feature maps were halved from layer to layer. From (7), this doubles the output dimension after each transposed convolutional layer. The process is repeated until the output has the desired dimension of $[64, 64, 3]$.

D follows the inverse operation. It begins by accepting inputs in the shape of $[batch_size, 64, 64, 3]$, where $batch_size$ was initialized to 128 during experimentation. The input is then passed through a series of strided convolutional layers, which decrease the size according to:

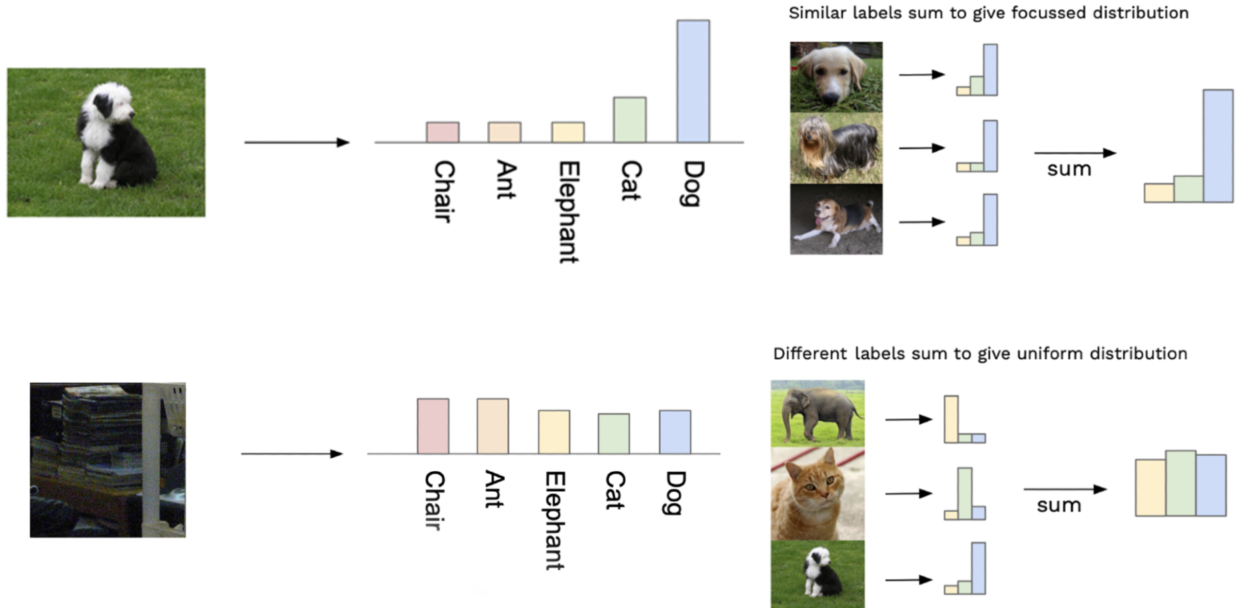

Fig. 2. Example probability distribution produced by InceptionV3

$$D_{out} = \frac{D_{in} + 2 * P - (K - 1) - 1}{S} \tag{8}$$

The hidden layers were all initialized to halve the image dimension while doubling the number of feature maps. From (8), this works out to values: $S = 2, P = 1, K = 4$. This process was repeated until the output has a dimension of $[2, 2, 1]$. This output was then passed to a Sigmoid activation layer for binary classification.

III. Results

As proposed in [1], discriminator pretraining was implemented to maintain healthy competition between G and D by allowing the discriminator to perform k optimization steps before updating the generator. Each experiment was conducted on a batch of 50000 images from CelebA for a total of 20 epochs. Detailed results can be viewed in **Appendix C.**

It was found that when $k = 1$, the generator and discriminator converge rather quickly. This means no further training would improve the system, however, the FID score is rather high, prompting some improvement to D. This can be done either by adjusting the architecture and/or hyperparameters or allowing for pretraining. Setting $k = 2$ and maintaining the same architecture does not result in much change in the FID score after 20 epochs, but it is apparent by the graph that the networks may have not yet converged. Further training could improve the quality of the synthesized images. Lastly, setting $k = 3$ resulted in the Discriminator becoming too powerful, collapsing the generator after only 10 epochs.

IV. Future Works

There are many ways to improve the images generated by the DCGAN. A technique called *progressive growth* was introduced by [6] in 2017. This training method begins by generating very small 4×4 images and gradually adding more layers to increase the size to 8×8, 16×16, and so on. The advantage to this is it allows the generator to maintain stability by first learning low-level feature such as facial structure and color. As the network grows, G is then able to focus more on finer details. There are also a number of normalization techniques that can be employed to aid in stabilization and improve results. Also introduced by [6], pixel-wise feature vector normalization normalizes the feature vector associated with each individual pixel to the unit length after every convolutional layer.

In addition to improving results by modifying the network and/or training loop, a more robust performance evaluation system would benefit the project greatly. FID scores help give insight into the quality and diversity of generated images without the need for visual inspection, but it does not tell much about the convergence of the GAN. The loss graph can be useful in determining whether the networks have converged, but it is not very reliable. In the future a method for measuring the distance G's probability distribution, p_g, mentioned in Section **I. Background** is to D's probability distribution p_{real}, would provide a useful metric when evaluating convergence [1]. This is important because it will help conclude whether the network has yet to converge, successfully converged, or even diverged from a previous convergence.

REFERENCES

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, “Generative Adversarial Nets,” NIPS’14: *Proceedings of the 27th International Conference on Neural Information processing Systems*, vol. 2, pp. 2672, 2680, Dec. 2014. Accessed: May. 4, 2022. doi: 10.1145/3422622. [Online]. Available: https://dl.acm.org/doi/10.1145/3422622

[2] J. Hui, “GAN - Ways to Improve GAN Performance,” *towardsdatascience*, 19-Jun-2018. [Online]. Available: https://towardsdatascience.com/gan-ways-to-improve-gan-performance-acf37f9f59b. [Accessed: 02-May-2022].

[3] “GAN Training,” *developers.google.com*. [Online]. Available: https://developers.google.com/machine-learning/gan/training. [Accessed: 01-May-2022].

[4] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium, “ NIPS’17: *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Dec. 2017 Accessed: May 5, 2022. doi: 10.5555/3295222.3295408. [Online]. Available: https://dl.acm.org/doi/10.5555/3295222.3295408

[5] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[6] T. Karras, T. Aila, S. Laine, and J. Lahtinen, “Progressive Growth of GANs for Improved Quality, Stability, and Variation”, *The 6th International Conference for Learning Representations*, 2018.

[7] D. Mack, ”A Simple Explanation of Inception Score”, *Medium*, 2019.

APPENDIX A.

Derivation of GAN value function from Binary Cross Entropy (BCE)

Given the formula for Binary Cross Entropy (BCE),

$$BCE = \sum_{i=1}^{N} y_i * log(p(y_i)) + (1 - y_i) * log(1 - p(y_i))$$

Where:
y_i = Label of element i
$p(y_i)$ = Predicted label of element i

Let:
x = a real image
z = random noise
$G(z)$ = A synthetic image generate by G from noise z
L = Loss value

The total loss for D can be computed as $L_D = BCE[D(x)] + BCE[D(G(z))]$. We will consider each operand individually.

Consider $BCE[D(x)]$.
Choose y_i = 1, since we aim to predict positive (real) samples

$$\begin{aligned} BCE[D(x)] &= y_i * log(D(x)) + (1 - y_i) * log(1 - D(x)) \\ &= 1 * log(D(x)) + (1 - 1) * log(1 - D(x)) \\ &= log(D(x)) \end{aligned} \tag{9}$$

Consider $BCE[D(G(z))]$.
Choose y_i = 0, since we aim to predict negative (fake) samples

$$\begin{aligned} BCE[D(G(z))] &= y_i * log(D(G(z))) + (1 - y_i) * log(1 - D(G(z))) \\ &= 0 * log(D(x)) + (1 - 0) * log(1 - D(x)) \\ &= log(1 - D(G(z))) \end{aligned} \tag{10}$$

We can now simply add (9) and (10) to arrive to the following value equation for D:

$$L_D = log(D(x)) + log(1 - D(G(z))) \tag{11}$$

WLOG, the loss function for G, L_G can be derived by only considering the rightmost operand:

$$L_G = log(1 - D(G(z))) \tag{12}$$

■

APPENDIX B.

PyTorch implementation of minibatch discrimination

```
class MinibatchStddev(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        #Compute stddev of all feature maps
        fmap_mean = torch.mean(x)
        sq = torch.square(torch.mean(x - fmap_mean))
        stddev = torch.sqrt(torch.mean(sq))

        #Calculate mean of stddev maps
        avg_stddev = torch.mean(stddev)
        shape = x.size()
        minibatch = torch.tile(avg_stddev, dims=[shape[0], 1, shape[2], shape[3]])

        #cat to input
        return torch.cat([x, minibatch], dim=1)
```

Fig. 3. PyTorch implementation of minibatch standard deviation

Appendix C.
Generation results

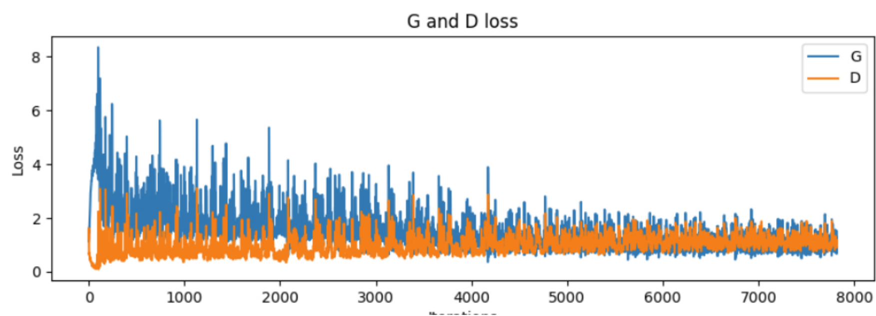

Fig. 4. Output at epoch 20, k=1, FID = 9.616

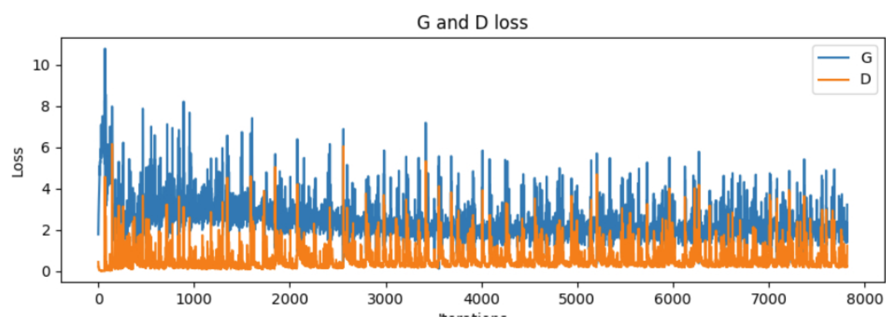

Fig. 5. Output at epoch 20, k=2, FID = 9.502

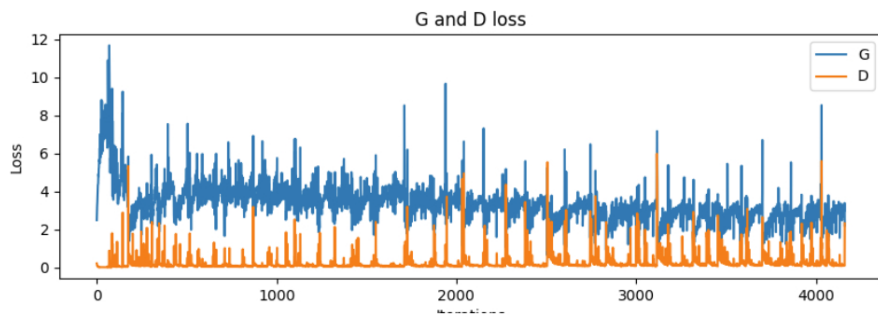

Fig. 6. Output at epoch 20, k=3, FID = NaN