IMA206

-

Project Report

SinGAN: Learning a Generative Model from a Single Image

Under the supervision of Arthur LECLAIRE *and* Yann GOUSSEAU

William LIAW
Anastasiia KARPOVA
Valentin ABRIBAT

2023-2024

Contents

1 Introduction **3**
- 1.1 The SinGAN model . . . 3
- 1.2 Objectives . . . 3
- 1.3 Understanding SinGAN . . . 3

2 Method **5**
- 2.1 Implementing the model . . . 5
  - 2.1.1 Model architecture . . . 5
  - 2.1.2 Fixing the training functions . . . 6
  - 2.1.3 Making the code more readable . . . 6
- 2.2 Defining the hyper parameters . . . 6
  - 2.2.1 Scale factor . . . 6
  - 2.2.2 Number of epochs to train per scale . . . 7
  - 2.2.3 Gsteps and Dsteps . . . 7
- 2.3 Loss function . . . 7
- 2.4 Padding . . . 8

3 Results **9**
- 3.1 Image generation: qualitative analysis . . . 9
- 3.2 Fréchet loss: qualitative analysis . . . 10
- 3.3 The SIFID metric: quantitative analysis . . . 11
- 3.4 Different paddings: qualitative analysis . . . 12

4 Problems encountered **14**

5 Conclusion **14**

6 Bibliography **14**

1 Introduction

1.1 The SinGAN model

In the recent years, being able to mimic the the internal distribution of patches has demonstrated itself as a very important tool in computer vision, mostly because it helps with classical problems such as denoising, deblurring, dehazing or image editing. To fulfill this need, a considerable amount of algorithms were developed using Generative Adverserial Networks (GANs), since this method proved to be effective to generate realistic results.

However, every method used to generate images was confronted to the same problem: generating realistic patches and textures required massive databases and consequently great computational power. This is an issue, as image generation through patch reproduction is particularly useful with limited datasets at disposal, for example in the case of rare biological malformations. Aware of this problem, a team led by Tamar Rott Shaham introduced SinGAN in 2019, a groundbreaking method that allowed to train a GAN on a single real image.

What allowed SinGAN to be effective was its use of a pyramidal network of GANs, each working at a different scale of the image which allows the learning of various image features, while keeping the coherence of the global image. With these innovations, SinGAN proved itself to be not only very effective, but also introduced new possibilities such as generating images of varied resolutions with the same training image.

1.2 Objectives

The goal of our work was two-fold: first, implementing the SinGAN algorithm in order to deeply understand its architecture and to be able to generate images with modified hyperparameters; the second part aimed at modifying the loss function of the model in order to examine if we could accelerate the training of the model with simpler affine assumptions. Additionally, we aimed to experiment with different padding functions to examine their effects on the images, most noticeably on their borders.

As SinGAN a model with a non-negligible complexity, an important part of our work was dedicated to understanding its core functionalities. This next section is thus dedicated to a brief explanation of the ideas behind it.

1.3 Understanding SinGAN

SinGAN is an unconditional generative model capable of generating images from random noise, uniquely requiring only a single natural image for training. Unlike traditional GANs that rely on extensive datasets, SinGAN utilizes patches from the input image. By training on these various patches, SinGAN effectively learns both the texture and intricate patterns of the image, capturing its detailed characteristics and enabling the generation of diverse and coherent images.

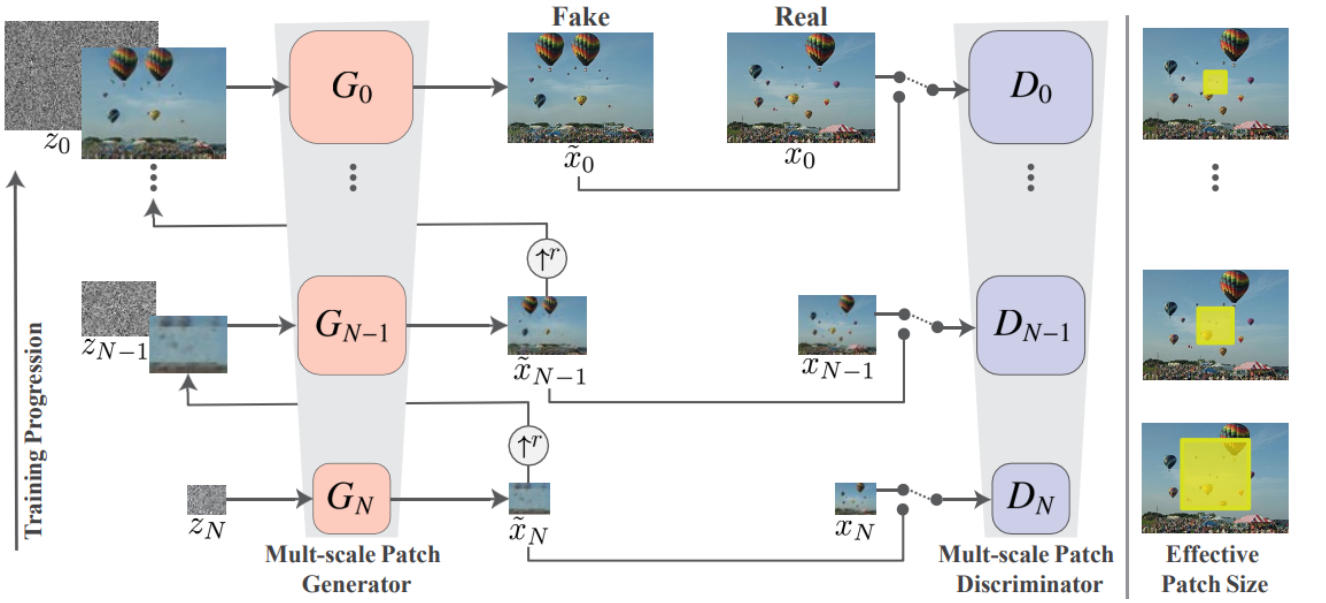

Figure 1: At each level, the generator $G_N - 1$ receives as an input the image $\tilde{x}_n$ generated by G_N, up-sampled at the current resolution (except at scale 0, where the Generator only receives random noise), and z_n a random noise image. Each output of the generator G_n is then sent to the discriminator D_n that will compare it to the original image at that resolution to compute the adversarial loss. It is important to note that the generation process at level n involves all generators $G_N, ...G_n$ and noise maps $z_N, ..., z_n$. This simplified figure was extracted from the original article.

The core concept of SinGAN involves constructing a pyramid of GANs rather than a single network. Each GAN in the pyramid is trained on a progressively subsampled version of the original image, as illustrated in **Figure 1**. Once a GAN is trained, the generator's parameters are fixed and it continues to influence the training of the subsequent GAN in the pyramid. This approach is crucial for capturing visual features at different scales, enabling the model to generate images that closely resemble the original in overall composition while introducing substantial variation in the finer details.

Delving deeper, each scale in the SinGAN pyramid operates as an individual GAN. At each scale, the generator takes a combination of noise and upsampled images produced by the previous generators (or only noise at the first scale) and outputs an image designed to deceive the discriminator. Each of SinGAN's discriminators, in turn, evaluates whether the generated image patches are real or fake. This configuration guides the generator to produce images that are progressively more realistic, capturing the intricate details and variations of the input image across different scales.

Thus, SinGAN can be used to solve various image manipulation tasks such as paint-to-image, editing, harmonization, super-resolution, and animation from a single image using just one generative network without additional training or information.

2 Method

In this section, we outline the steps taken to achieve our objectives. Initially, we adapted the source code associated with the article to ensure it was functional, and we refactored certain components to modify its behavior. Next, we discuss the hyperparameters crucial to our work, detailing our selection and adjustments. Following this, we delve into the specifics of the adversarial network's loss function, highlighting the modifications we made. Finally, we describe the alterations applied to the padding technique to investigate border effects in image generation.

2.1 Implementing the model

Despite the availability of an official implementation accompanying the original article, we faced several challenges while attempting to use it. The code was highly modular and convoluted, making it difficult to understand and modify. For example, some variables were reused for vastly different purposes and changed sizes multiple times within a few lines of code, with minimal explanations. Additionally, the parameters accessible to and necessary for each function were majorly unclear. The specified versions of PyTorch were outdated, leading to incompatibility issues that required updates to newer versions, specially since the official implementation does not present a list of requirements. Due to these complexities, for this project, we opted to re-implement the code pertinent to our objectives in a Jupyter notebook, which you will find attached to our submission.

2.1.1 Model architecture

Since we wanted to focus on modifying the hyper parameters and the loss function rather than modifying the underlying architecture, we chose to keep the **ConvBlock** sequential class that was described in the article:

$$\text{ConvBlock}(in\_channel, out\_channel, ker\_size, padd, stride) \tag{1}$$

- **Convolutional Layer**: This is the classic layer that we've seen many times, it is responsible for feature extraction by applying a set of filters (kernels) to the input image.
- **Batch Normalization**: This layer normalizes the activations of the previous layer at each batch, maintaining the mean activation close to 0 and the activation standard deviation close to 1 to prevent instability.
- **LeakyReLU Activation**: Introduces non-linearity to the network and helps to avoid the vanishing gradient problem.

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.2x & \text{if } x < 0 \end{cases} \tag{2}$$

Both the Generator and the Discriminator are constructed using a modular block structure. Each consists of a head composed of a single **ConvBlock**, a body containing a user-defined number of **ConvBlocks** (typically three), and a tail. For the Generator, the tail includes a **Conv2D** layer

followed by a hyperbolic tangent activation function to normalize images within the range [-1, 1]. The Generator also incorporates a skip connection that can either bypass the input image or noise and adds it to the output of the **ConvBlocks**. In contrast, the Discriminator's tail outputs a singular channel per patch, effectively providing a feature per patch on the image.

We retained the original weight initialization parameters since experiments revealed no significant performance differences with alternative initialization schemes.

2.1.2 Fixing the training functions

We identified an issue with the original code's `train_single_scale` function: the fake image appeared to be incorrectly defined during the generator network's updates at each iteration. Specifically, for a given epoch, the training of the generator utilized the same fake image, the last generated from the discriminator's training loop. To resolve this, we added a condition to effectively produce a new fake image: If it was the first generator, we used only noise to construct the fake image `prev`; otherwise, we employed the `draw_concat` function to incorporate the outputs from previous scales. This modification ensured that `prev` was appropriately constructed based on the current scale, leading to correct updates in the G network.

2.1.3 Making the code more readable

We encountered difficulties understanding the code, primarily due to frequent instances of variables being reused and modified without clear explanations. Additionally, some functions could benefit from direct optimization. For example, we opted to remove the original **create_reals_pyramid** function and integrated its functionality directly into our pipeline. Furthermore, the scope and return values of the original functions were often unclear: sometimes they returned standard values, other times they returned by reference, and in some cases, they updated global dictionaries.

2.2 Defining the hyper parameters

The main hyperparameters we used to train the SinGAN model were: pyramid scale factor (**scale factor**), number of epochs to train per scale (**niter**), **Gsteps** and **Dsteps**. By varying the values of these parameters, we adjusted the balance between the model's performance and the resources necessary for its training.

2.2.1 Scale factor

A larger scale factor results in a pyramid with more levels, enabling the model to progressively capture finer details and thereby generate more detailed images. However, this also increases computational complexity, and occasionally artifacts may appear in the generated images. Conversely, a smaller scale factor allows the model to concentrate on broader structures, making it more computationally efficient. However, finer details may be overlooked, resulting in simpler images.

Thus, after conducting a series of experiments with different values of the scale factor, we chose a scale factor equal to **0.75**. Using this value, the model requires less computational resources, but

successfully captures both fine detail and overall structure, generating detailed but not cluttered images.

2.2.2 Number of epochs to train per scale

After setting the value of the scale factor parameter to 0.75, we conducted a series of experiments to determine the optimal value of the number of epochs to train per scale. In the original code, the authors had used the default value of 2000. By reducing the parameter value to 1000, we were still able to obtain realistic images and a low quantitative metric value, while also reducing the training time. Therefore, the value of 1000 was chosen as optimal, allowing to obtain of satisfactory results in a reasonable time.

2.2.3 Gsteps and Dsteps

These parameters specify how many times the generator's and the discriminator's parameters are updated at each iteration. Typically, in the original code, this value is set to 3 to strengthen the adversarial learning between these two components. However, our initial implementation failed when 'Gsteps' was set to any value other than 1. Eventually, after resolving the issue, we observed comparable image generation results, but a much greater necessary execution time when both parameters were set to 3. Consequently, we determined that maintaining values of 1 for 'Gsteps' and 3 for 'Dsteps' respectively yielded the best empirical results.

2.3 Loss function

When working with adversarial networks, the loss used to train the model consists of two distinct terms: the adversarial term and the reconstruction term. This distinction is crucial because it involves training two processes simultaneously: enhancing the generator's capability to create fake images indistinguishable from real ones, and training the discriminator to distinguish these fake images. In the original paper, the authors advocate for using the Wasserstein loss [2] with gradient penalty to improve training stability. Therefore, we initially implemented the loss function based on this method.

In our train model, we start by training the discriminator to maximize $D(x) + D(G(\tilde{x}))$, with $\tilde{x}$ being the combination of the previous outputted image and the new noise. This is because D(x) gives the probability of x being real, with 0 being the result for a certified fake and 1 a real image, so since we are using the WGAN-GP loss, this makes sens because $D(x)$ will have a value progressively evolving towards 1 and $D(G(\tilde{x}))$ will slowly be equal to 0, preserving the sum to a value near 1 which gives more stability. After this first step, we then maximize only $D(G(\tilde{x}))$ in order to retrieve the reconstruction term: this will help our generator learn which patch features are the most effective to trick the discriminator, and will consequently enhance its capacity to create realistic images.

Once both terms have been calculated, and each GAN has been trained, they are kept fixed and we can compute our training loss:

$$\min_{G_n} \max_{D_n} \mathcal{L}_{adv}(G_n, D_n) + \alpha \mathcal{L}_{rec}(G_n)$$

Once we had coded this whole part, we were ready to generate fake images as described in the paper. This is when we realized that the biggest drawback to this method was the long training time: using GPUs, the time it took to train a single image model could go up to one hour and a half when using the base parameters. Consequently, it was this point that we began the second half of our work, which was to simplify this method.

In line with our project objectives, we identified an alternative approach. Here, GANs collaborate across different scales to emulate the internal distribution of patches. Thus, we opted to eliminate the discriminator and replace it with an affine method. To achieve this, we developed a function to manually extract patches (maintaining an effective patch size of $s = 11$) from both the real and generated images at each level. These patches were organized into matrices where each column corresponds to a patch, enabling the calculation of their mean and covariance. Subsequently, we chose to utilize the Fréchet Distance, without relying on the typical Inception model, to compute the loss:

$$\text{FID} = \|\mu_{real} - \mu_{fake}\|^2 + \text{Tr}(\Sigma_{real} + \Sigma_{fake} - 2(\Sigma_{real}\Sigma_{fake})^{1/2})$$

With this method, we hypothesized that we could nearly half the training time, the discriminator no longer needing any training, but we also expected worse qualitative results because a simple affine transformation does not allow deep features to emerge.

2.4 Padding

In the context of convolutional neural networks, padding is used to preserve the spatial dimensions of the input image following convolution operations on a feature map. This process involves adding extra pixels around the border of the input feature map before convolution. There are various padding techniques, the most common of which is zero padding (**ZeroPad2d**), which involves the insertion of zeros at the borders of the input feature map. This is the method employed by the authors in the original code to pad the noise tensors and scaled images during model training.

We formulated a hypothesis that padding with values other than zero, such as the mean of pixels near image boundaries, could improve model results and allow the use of information from the edges of the input images during the convolution process. To test this hypothesis, the **ConstantPad2d** module was used, which substitutes a constant value into the input tensor boundaries. To determine the optimal value of the constant, a function was created that takes an image as input, extracts boundary pixels from it, and returns their average value.

As an experiment, we also tested other methods of padding, which used initial values of the images: **ReplicationPad2d** and **CircularPad2d**. In the **ReplicationPad2d** module, the input tensor boundaries are replicated, whereas in the **CircularPad2d** module, the input tensor boundaries are circularly padded.

3 Results

3.1 Image generation: qualitative analysis

We began by generating images using parameters nearly identical to those in the original GitHub repository to assess the quality of our implementation. The only deviation was setting the **Gsteps** variable, the number of iterations on the training loop of the generator, to 1, as this produced superior results compared to the default value of 3, a comparable generated image quality, with less computing time. As illustrated in **Figure 2**, our results closely resemble those presented in the original paper. Notably, the visual features —such as birds and trees in the background—are generated throughout the image with considerable coherence.

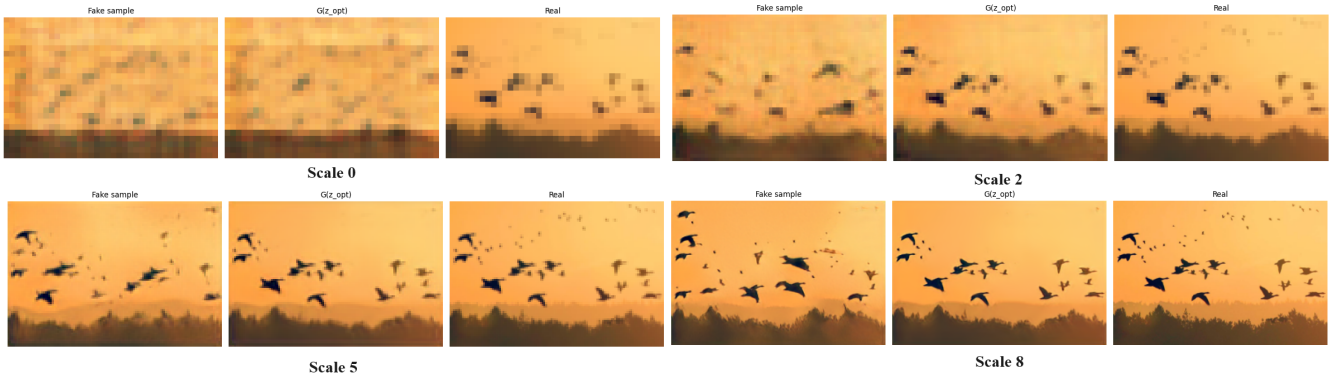

Figure 2: From left to right: 4 sets of three images, the first being randomly generated image using the trained generator, the second being a generated image using the optimal noise, and the third being the real image. The scales from which each set has been extracted are: 0, 2, 5 and 8.

To verify if our Random Samples method functioned too, we displayed randomly selected outputs with the same settings as mentioned before. Once again, we successfully obtained results similar to what was described in the original paper, with, in this case, birds appearing a various places and having different shapes and sizes. On this example, that we exhaustively analyzed, we remarked that some patches appeared in virtually every random generation, for example here the bird near the top left of the image, who sees little to no variation at each generation.

Figure 3: Samples randomly generated from the previous model

After this, we also decided to plot the evolution of each of our losses for both the generator and the discriminator to try to understand how it evolved at each step. We can observe the results in **Figure 4**:

Figure 4: From left to right: 4 sets of 2 images, the first each time being a graph of the Generator loss and the reconstruction loss, the second being a graph of the Discriminator loss (real, fake and total).

As we can see on these graphs, the reconstruction loss always quickly converges towards 0, which means that our model is able to generate images that correspond to the reality. However, our generation loss, which is basically calculated but the generator's ability to trick the discriminator is disappointingly constant at each scale. We hypothesize that we get these results because we chose to set **Gsteps** to 1, which means that the generator does not refresh nearly as much as in the original model. The solution to this would be to fix the issues we had when setting **Gsteps** value to anything other than 1, since it would make the model more dynamic.

3.2 Fréchet loss: qualitative analysis

In order to evaluate whether or not our replacement loss function was effective to simulate the original SinGAN model, we use the exact same pipeline as for our qualitative analysis on our original implementation with unchanged parameters and we compare the results in **Figure 5**.

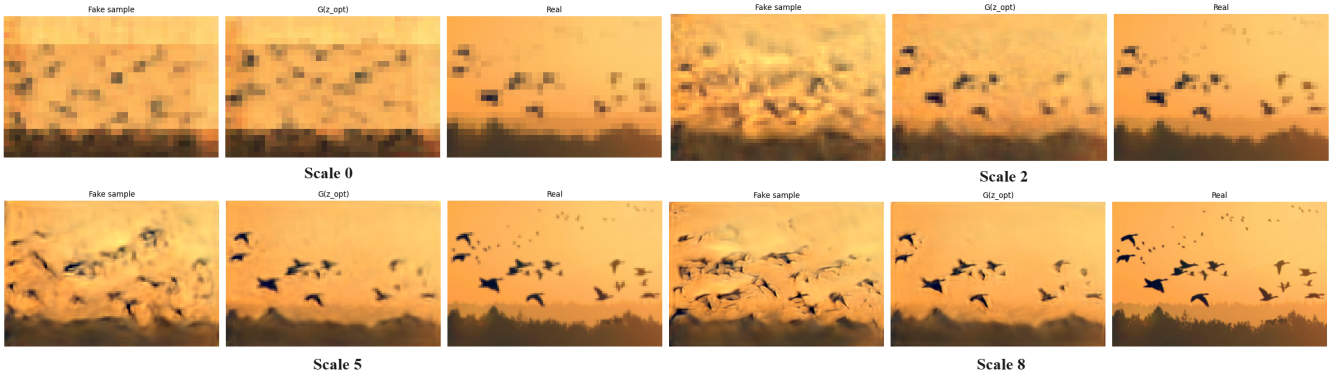

Figure 5: Same as figure 3, to read by sets of 3 images going from left to right. Instead of using the original loss, the model here was trained with our own Fréchet loss.

As we can easily see, this loss function is way less effective at generating shapes that resemble

anything that could be realistic. This is easily understandable, because the limit of our new method is that we simply compute the mean and the covariance of each patch and compare it to the ones of the original image. Because of this, the pyramidal architecture only helps the generator progressively understand which areas of the image should have what intensity, but it is completely unable to learn precise shapes. This is highlighted in **Figure 6**: since the loss is only computed with a simple comparison of the mean and the covariance at different scales, the loss quickly converges towards 0, without it having an important effect on our image.

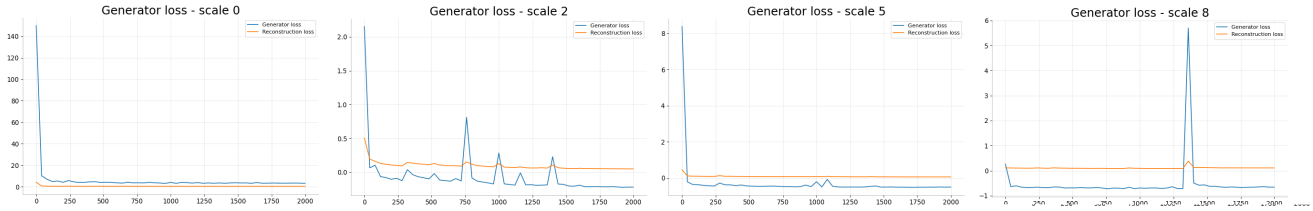

Figure 6: From left to right: graphs of the evolution of our generator loss through the iterations for the scales 0, 2, 5 and 8

.

However, as we had theorized in our Method section, using the Fréchet is way faster than the original code. As explained before, this is due to the absent need to train the Discriminator, and, since we replace its role with affine operations only, it is less computationally costly.

Table 1: Average time to train the model (in minutes)

Number of iterations per scale	**Original Loss**	**Fréchet Loss**
1000	28	16
2000	60	31

3.3 The SIFID metric: quantitative analysis

In order to verify if their results were realistic, the original SinGAN authors used two methods: the first was to simulate a mechanical turk, by simply asking people if they thought that the generated images were real or not, the second was to use a well known metric, the Single Image Fréchet Inception Distance (SIFID), which was the method that we decided to implement in our code.

The basic idea behind the base FID metric, on which SIFID is based, is that we calculate the resemblance between two images by passing them through a pre-trained Inception Network that extracts the high-level features of both images, and then compares them using the Fréchet distance that we've introduced earlier. SIFID is a bit different because we only have a single real image and we are more interested in its internal patch statistics, so instead of using the activation vector after the last pooling layer, the research team chose to extract the deep features at the output of the convolutional layer juste before the second pooling layer.

The SinGAN team adapted the FID metric to a single image using the average score of 50 images for full generation. In the SIFID method we implemented, we calculated the FID for one real image used multiple times and 50 randomly generated samples. To calculate the FID, we used the **Fréchet Distance** method implemented in **torchaudio.functional**.

The SIFID values obtained when generating birds.png images are presented in **Table 2**. Despite the fact that the obtained values are significantly smaller than the values presented in the article, their behavior corresponds to reality: we obtain smaller values for images that are more similar to real ones , and large SIFID values for images that differ significantly from the original ones.

Table 2: Received SIFID scores

	SIFID
Original Loss	2.1939146e-05
Fréchet Loss	1.1237572e-05
Original Loss + ConstantPad	1.2019249e-05
Original Loss + ReplicationPad	1.227873e-05
Original Loss + CircularPad2d	0.00030369556

3.4 Different paddings: qualitative analysis

The use of **ConstantPad2d** and **ReplicationPad2d** paddings produce results that are visually comparable to those obtained with **ZeroPad2d** (**Figure 7**). At the same time, the SIFID value for these methods decreases slightly: from 2.1939146e-05 to about 1.2019249e-05, while the training time increases by only 1-2 minutes.

Conversely, the utilization of **CircularPad2d** has been observed to significantly deteriorate the outcomes of the trained model: artifacts and distortions appear in the images (**Figure 7**), accompanied by an increase in the SIFID value to 0.00030369556. These outcomes can be attributed to the potential for circular padding to compromise the integrity of the image and introduce inappropriate elements.

Thus, using **ConstantPad2d** with a constant equal to the mean value of the image's border pixels, or **ReplicationPad2d**, which uses the values of the border pixels themselves for padding, is a worthy alternative to **ZeroPad2d**. These methods not only maintain the visual quality of the images, but also demonstrate a slight improvement in the model's performance.

Figure 7: From left to right: image obtained using ZeroPad2d, ConstantPad2d with the mean of the image's boundary pixels as a constant, ReplicationPad2d and CircularPad2d.

4 Problems encountered

Throughout this project, we encountered several issues that were initially overlooked. Initially, a significant amount of time was devoted to familiarizing ourselves with the code. Despite our theoretical understanding of the method, the original code was cluttered with numerous functions irrelevant to our objectives, lacked documentation, and had other previously noted issues. This situation considerably delayed our progress as we struggled to initiate certain parts of the code.

Another significant challenge that consumed a considerable amount of our time was accessing powerful GPUs. While some group members had reliable access to local GPUs, others relied on online platforms such as Google Colab and Kaggle for processing units. It's important to note that maintaining stable access to an online GPU, especially with platforms like Colab or Kaggle, was challenging. This was primarily due to the extensive image generation tasks we undertook, which frequently led to disconnections from the runtime. Each image generation task required substantial computation, often exceeding personal usage limitations.

5 Conclusion

To conclude our work on this project, we believe that we successfully achieved our primary objective of understanding the publication paper, the original code and implementing it on a compatible and more understandable notebook, making this method more accessible.

We also successfully implemented a new loss function that significantly reduced the training time for our models. Although the results were less optimal compared to using the original loss, this outcome was expected. We believe that future research could expand upon this initial finding by exploring semi-discrete optimal transport as a new loss function, rather than solely relying on a purely affine model. Additionally, we believe fine-tuning the training hyperparameters could lead to more favorable results. Moreover, we managed to achieve SIFID values close to those computed by the official implementation, although we encountered discrepancies compared to the results reported in the original article.

Finally, experimenting with different padding methods revealed that this parameter could significantly impact our model's performance without compromising the quality of generated images. We regret not having more time to explore a wider range of parameter combinations to further investigate how they could potentially enhance our results.

6 Bibliography

[1] Shaham, T. R., Dekel, T., & Michaeli, T. (2019). SinGAN: Learning a Generative Model from a Single Natural Image. arXiv. https://doi.org/10.48550/arxiv.1905.01164

[2] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein GANs. In Advances in Neural Information Processing Systems, pages 5767–5777, 2017