Universitat Politècnica de Catalunya
Instituto de Astrofísica de Canarias

Master in Artificial Intelligence

Final Thesis Project

Generation of Synthetic Solar Images with GANs

Author:
Jorge Sierra Acosta
Master in Artificial Intelligence, UPC

Supervisor:
Josep Ramon Morros
Signal Theory and Communications Department, UPC

Co-supervisor:
Andrés Asensio Ramos
Instituto de Astrofísica de Canarias (IAC)

Tutor:
Nuria Castell Ariño
Computer Science Department, UPC

January 31, 2020

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

ESCOLA TÈCNICA SUPERIOR
D'ENGINYERIA
Universitat Rovira i Virgili

UNIVERSITAT DE BARCELONA
Facultat de Matemàtiques i Informàtica

Abstract

This works presents an alternative solution to the already existing physical simulations for the Sun's photosphere which are used to output a single texture image from the Sun's surface. This simulations are slow and require huge computational power, a cheaper alternative in time and resources is introduced.

Architectures for a Generative Adversarial Network and a Variational Autoeconder are considered and trained in the generation of small image tiles (128x128px) that resemble the surface of the Sun around an area of approximately $2.62 * 10^7 km^2$ *which with the proposed method can be composed into an image of arbitrarily any size given enough tiles.*

Different architectures along with fine tuning is used to obtain the best network possible in both cases. Their results are compared, but the Generative Adversarial Network shows a powerful improvement on the generation of said tiles compared to the Variational Autoencoder.

Lastly some methodologies for stitching together the generated tiles are presented including a technique that uses a Genetic Algorithm approach to modify the generated tiles.

Contents

1 Introduction **3**
- 1.1 Objectives and scientific background . . . 3
- 1.2 Resources . . . 5
- 1.3 Artificial Intelligence Background: GANs . . . 5
- 1.4 Artificial Intelligence Background: VAEs . . . 6

2 Related Work **8**
- 2.1 DCGAN and general notes for GAN training . . . 8
- 2.2 Progressive growing of GANs . . . 9
- 2.3 Texture generation . . . 10
  - 2.3.1 Spatial GANs . . . 10
  - 2.3.2 Periodic Spatial GANs . . . 10
  - 2.3.3 TileGANs . . . 10

3 Generating synthetic solar images **12**
- 3.1 Dataset & data augmentation . . . 12
- 3.2 Generating synthetic solar images with VAEs . . . 14
- 3.3 Generating synthetic solar images with GANs . . . 15
- 3.4 Tile stitching . . . 21
  - 3.4.1 Blending similar tiles with a greedy search . . . 21
  - 3.4.2 Blending similar tiles with a greedy search in latent space 23
  - 3.4.3 Refining tiles with a GA . . . 23

4 Software **26**

5 Conclusions **28**

6 Future work **29**

7 Annex **33**
- 7.1 Example of some generated images . . . 33

1 Introduction

1.1 Objectives and scientific background

The main purpose of this project is to validate whether it's possible to generate synthetic solar images of any size resembling the "blobs" commonly seen in the texture of the real solar surface: the photosphere; namely following the same statistical spatial distribution, see figure 1. The granules seen on the photosphere of the Sun are caused by convection currents of plasma within the Sun's convective zone. The bright spots (green and yellow in figure 1) are located in the center of those granules where the plasma is hotter which then descends into the dark colder regions (blue and purple).

Note that sunspots, a temporary phenomena on the Sun's photosphere that appear as spots darker (regions of reduced temperature) than the surrounding areas, are not considered in this work, as they are a spontaneous phenomena caused by concentrations of magnetic field flux that inhibit convection currents.

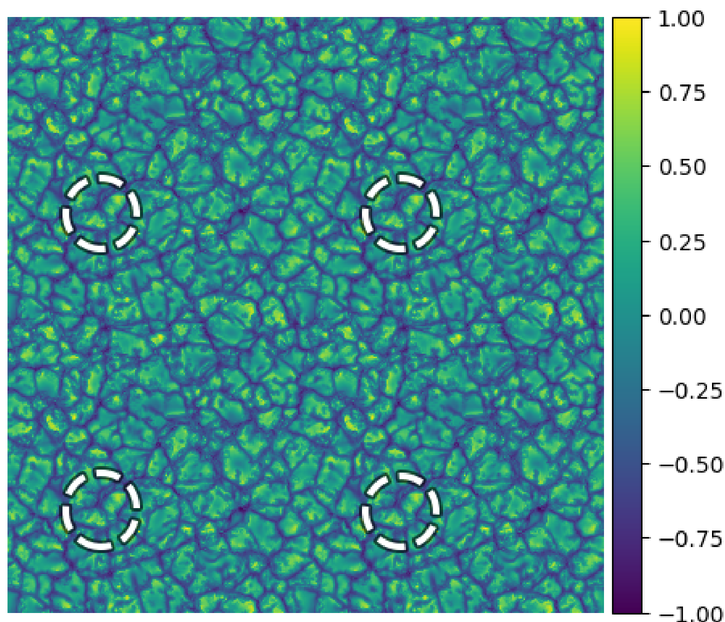


Figure 1: Composite image (3600x3600px) of four periodic tiles from the same image (1800x1800px) obtained with a physical simulation of the Sun's photosphere (telescope diameter is 4 meters). Some repeating patterns are marked with white circles. The composite image represents an area of approximately $1.3 * 10^9 km^2$ of the Sun's surface.

Normally when these images are not directly acquired using a real telescope they are generated using a physical simulation of the magnetohydrodynamics of the Sun's surface (magnetic properties and behaviour of electrically conducting fluids) often using periodic boundary conditions (PCBs) where the geometry of the 2D cells satisfy perfect 2D tiling, useful for large or infinite

systems, resulting in series of different images each one representing a different point in time that can be perfectly tiled to obtain an infinite repeating texture; however, this is not ideal in some cases. Regardless of this simplification, the simulation still requires a big amount of computational power. In this case the images used in this work where simulated in the *LaPalma*[1] supercomputer (table 1) which belongs to the Public Research Organization: ***Instituto de Astrofísica de Canarias*** or IAC for short, located in Spain (Canarias). Furthermore, the calculation of each one of the simulation steps takes several minutes in this machine.

LaPalma supercomputer	
Peak performance	83.85 TFlops
Main memory	8 TB
Number of nodes	4032
Main nodes	4032 cores Intel Xeon SandyBridge a 2.6Ghz
Type of CPU	Intel Xeon SandyBridge-EP E5-2670/1600 20M
Cores per node	8
GFlops per core	20.8 GFlops
Interconnection networks	Mellanox Infiniband FDR10

Table 1: Some of the technical specifications of the supercomputer used for the photosphere physical simulation.

Even though the composited texture resulting from the infinite tiling of a single image may trick the human eye, it's not good enough for other matters. Moreover, because this method is a simulation, it can not generate two different enough samples without generating all the middle states between both images. Thus the need to search for a better solution in the generation of these type of textures.

The main application of the new synthetic images will involve their usage as inputs for different telescope adaptive optic (AO and MCAO) simulations for solar observation. The classical adaptive optics methods used in telescopes employ a deformable mirror in order to correct the wavefront aberrations produced at different altitudes (atmospheric turbulences). They, however, have a narrow field of view, this problem is solved by the multi conjugate adaptive optics (MCAO) which uses more than one deformable mirror.

The IAC will employ the generated images as an inmediate first solution to train and validate a neural network model to control the deformable mirrors in the MCAO systems as the number of currently existing solar surface images is not enough for this purpose. This requires the images to be of a high resolution, around 9 MB or more. Analyzing the photosphere of the Sun has also a real scientific interest, as it can be used to help make better predictions of the space weather or how the internal processes of the Sun operate. That's

[1] https://www.res.es/en/res-sites/lapalma

why this work also involves the creation of a ready-to-use standalone software capable of generating such images without the need of technical knowledge.

1.2 Resources

For this work the resources available include access to the resources of the Image and Video Processing Group (GPI)[2] which is a research group of the Signal Theory and Communications department from the UPC. Nevertheless most of the processing was done locally in a computer with the specs given in the table 2 which demonstrate that the proposed methods can be doable on a normal desktop computer in a fair amount of time.

	Local desktop resources
CPU	Intel Core i7 6700 @ 3.40GHz
RAM	16GB 1066MHz DDR3
GPU	NVIDIA GeForce RTX 2060 6GB

Table 2

1.3 Artificial Intelligence Background: GANs

This problem lies in the area of texture generation, two different main generative models exist for these kind of problems: **Generative Adversarial Networks** and Variational Autoencoders. The GAN model was proposed by *Ian J. Goodfellow et al.* in 2014 [1] (see figure 2). They proposed a new framework for estimating generative models via an adversarial process. Two models are simultaneously trained: a generative model G that captures the data distribution, and a generative model D that estimates the probability that a sample came from the training data rather than G. This makes them capable of learning to mimic any distribution of data (image, audio, text, ...), as an example, a generator may learn the probability distribution of a big set of images, where each image represents a sample from an n-dimensional distribution and output completely new samples, similar to how one could approximate the normal distribution that best fits a pair of 1-dimensional set of samples and generate new ones. Generator and discriminator train is defined as a minmax game with the following objective function (eq 1) the models converge when they reach the Nash equilibrium which is the optimal point for the given function:

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

Generative image models are well studied and fall into two categories: parametric and nonparametric. The non-parametric models usually do matching from a database of existing images, often matching patches of images, and

[2] https://imatge.upc.edu/web/

have been used in **texture synthesis** [2], super-resolution [3] and in-painting [4].

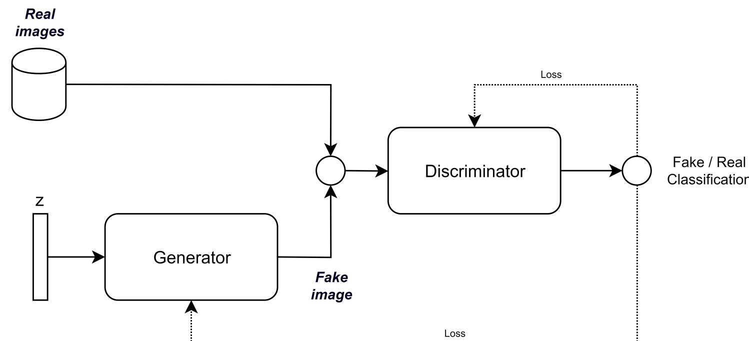


Figure 2: GAN model.

Note that the input given to the generative model is a random vector. The most common model for G is an inverse convolutional network which upsamples a random vector into an image. GANs are not trivial to work with, their most common problems are:

- Non-convergence: the model parameters oscillate, destabilize and never converge.
- Mode collapse: the generator collapses which produces limited varieties of samples.
- Diminished gradient: the discriminator gets so successful that the generator gradient vanishes and learns nothing.
- Unbalance between the generator and discriminator causing overfitting.
- Highly sensitive to the hyperparameter selections.

1.4 Artificial Intelligence Background: VAEs

Autoencoders (not Variational Autoencoders) are composed of an encoder which encodes the input data as a latent vector of dimensionality N that can be seen as a representation of the original data and a decoder, which transforms back this latent vector into the original data. These two models usually have a mirror architecture from one another. As an example, in a trained autoencoder for animal images and an ideal case, one may be find that the autoencoder is codifing specific variables into the vector such as, hair color, height, number of legs, ... Which are then reconstructed into the original image by the decoder.

However, autoencoders are not generally considered generative models because after training, a random latent vector may not be decoded into an actual image but only into noise (generalized denoising auto-encoders [5] may be considered generative models as they capture the structure of the original data distribution). Variational Autoencoders (VAE) (figure 3) on the other hand add an

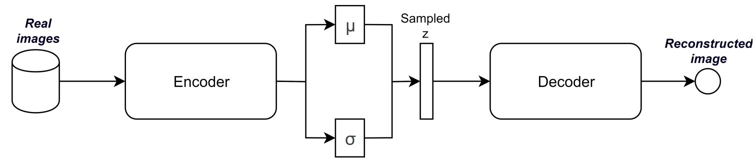


Figure 3: VAE model.

additional constraint to the encoder: the latent vector representation must follow a unit gaussian distribution.

One of the downside of the VAEs against GANs is that they produce more blurry images (figure 4) in comparison (however there are ways to avoid these problems [6]).

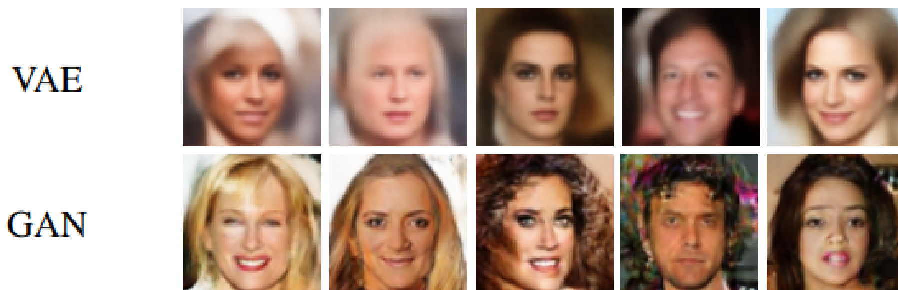


Figure 4: Comparison between images generated by a VAE and a GAN model from [7].

2 Related Work

Many techniques exists nowadays for the generation of textures (texture synthesis) with the most modern methods employing almost exclusively GANs. Around this topic, general information for improved GAN training has also been published and the model has already been used for astronomical related image generation [8].

2.1 DCGAN and general notes for GAN training

Deep Convolutional GANs: DCGANs [9] are one of the most common types of GANs which take advantage of the success of CNNs for image generative models. Some architecture guidelines for stable Deep Convolutional GANs are presented:

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator. It's recommended to not apply batchnorm to the generator output layer and the discriminator input layer which often result in model oscillation.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU [10] [11] activation in the discriminator for all layers to prevent the dying ReLu problem (ReLu units stuck in the negative side, always output 0).
- Recommended a learning rate of 0.0002 and a β_1 (momentum) of 0.5.
- All weights were initialized from a zero-centered Normal distribution with standard deviation 0.02.

The DCGAN model has been shown to have a good scale factor with increased data and image resolution [12]. Recent analysis has shown that there is a direct link between how fast models learn and their generalization performance [13]. Improvements to decrease the likelihood of the generator memorizing input examples a de-noising dropout regularized ReLu. To further decrease the likelihood of the generator memorizing input examples it is advised to use some deduplication preprocessing step for the input dataset [12].

However there are still some forms of model instability remaining: as models are trained longer, they sometimes collapse a subset of filters to a single oscillating mode [12].

It's common for CNN architectures to alternate convolutions with max-pooling layers followed by a small number of fully connected layers. On the other hand,

max-pooling can simply be replaced by a convolutional layer with increased stride without loss in accuracy on several image recognition benchmarks [14]. However as seen in figure 5, checkerboard artifacts may be formed when the stride of the convolution does not favor a good distribution of the kernel over the output.

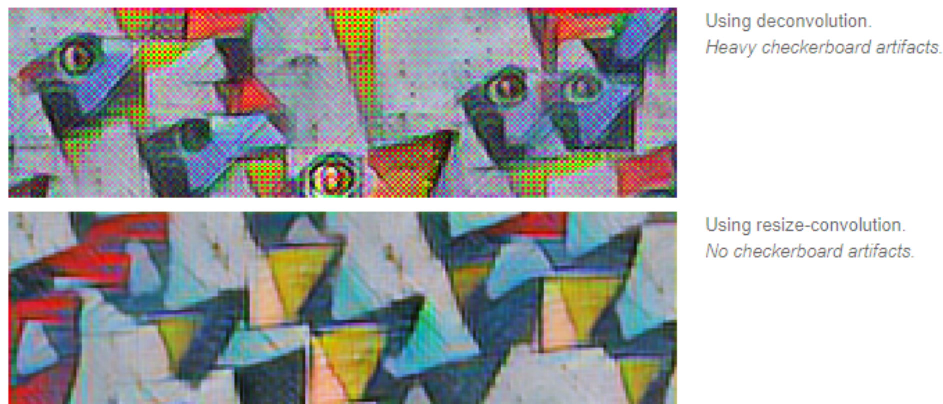


Figure 5: Checkerboard artifacts in deconvolution [15].

2.2 Progressive growing of GANs

It is presented a new method for training GANs where new blocks are dynamically added during training to increase the resolution of the output image once the generator has learned to generate good enough images at the current resolution (PGGAN) [16]. There are multiple details that are defined in this work that can be applied to GANs in general.

GANs have a tendency to capture only a subset of the variation found in training data, "minibatch discrimination" was suggested [17] as a solution where they compute feature statistics not only from individual images but also across the minibatch. This new layer learns a large tensor that projects the input activation to an array of statistics. PGGANs solution does not include learnable parameters nor hyperparameters. First compute the standard deviation for each feature in each spatial location over the minibatch. Then average these estimates over all features and spatial locations to arrive at a single value. Replicate the value and concatenate it to all spatial locations and over the minibatch, yielding one additional (constant) feature map. This new layer works better if inserted towards the end of the discriminator.

PGGAN have shown that batch normalization does not produce any real improvement in GANs training and the only constraint GANs need is controlling signal magnitudes and competition, for this, equalized learning rate is used, where a trivial $\mathcal{N}(0, 1)$ initialization is employed and weights are scaled at runtime. To disallow the scenario where the magnitudes in the generator and discriminator spiral out of control as a result of competition, the feature vector

is normalized in each pixel to unit length in the generator after each convolutional layer using a variant of "local response normalization" [18].

2.3 Texture generation

Older texture generation techniques included graph cuts. Patch regions from a sample image or video are transformed and copied to the output and then stitched together along optimal seams to generate a new (and typically larger) output [19]. In contrast to other techniques, the size of the patch is not chosen a-priori, but instead a graph cut technique is used to determine the optimal patch region for any given offset between the input and output texture. Eventhough this technique is used to generate a new image from a sample or a variety of samples from a texture it can be used to stitch together tiles of images that are generated by other methods.

VAEs have also been used for texture synthesis [20], nonetheless they are not commonly used for this purpose nor do they stand out from the results of other alternatives.

2.3.1 Spatial GANs

The key insight we had in SGANs [21] is that in texture images, the appearance is the same everywhere. Hence, a texture generation network needs to reproduce the data statistics only locally, and, when we ignore alignment issues, it can generate far-away regions of an image independent of each other. In a fully-convolutional network, a model can be trained on a certain image size, but then rolled-out to a much larger size. The resulting images locally resemble the texture of the original image on which the model was trained.

2.3.2 Periodic Spatial GANs

The first shortcoming of SGANs is that they always sample from the same statistical process, which means that after they're trained, they always produce the same texture unless they are trained in a set of images, resulting in an output which mixes the inputs.

The solution to this using PSGANs [22] involves having some global information that allows the model to differentiate what output shall it produce. This was achieved by setting a few dimensions of each vector in the spatial field of vectors Z to be identical across all positions instead of randomly sampling it as in the previous section (these dimensions are hence globally identical).

2.3.3 TileGANs

One of the models that most closely resembles the objective in this work is the TileGAN [23]: many any input images are given and a large-scale output is required. Using d progressive growing of GANs, already referenced before, they built a network that generates tile samples of different texture types and

propose a novel algorithm to combine the given outputs also allowing artistic control for the user. The generator is defined as (eq 2):

$$G(z) = G_{B_l}(G_{A_l}(z)) \tag{2}$$

Where z is a randomly sampled latent vector and l specifies the intermediate level at which latent field synthesis is performed. This synthesis consist of merging the latent feature maps information of different tiles to obtain a smooth transition in the upper layers of the network. Their approach generates a huge number of samples that are then downscaled and compared to what the user wants at a given location, a close enough match is found in the low resolution domain and then tiles are merged.

3 Generating synthetic solar images

3.1 Dataset & data augmentation

The number of images needed to train a GAN network varies depending on the task at hand and can only be estimated experimentally. Naturally, a huge number of images will be expected to produce a better result while training, however there might be a point where increasing input size will not improve the network in any significative way. This ideal quantity also depends on the generator and discriminator and generator architecture (e.g. batch normalization may reduce the number of examples needed), the desired output resolution, the latent vector size, the problem or image complexity, number of labels, etc... Some examples of previous works include the following datasets:

- MNIST [24] 60,000 examples for training and 10,000 for testing with 10 different hand written digit number types (0-9).
- Toronto Face Database [25]: around 3,000 samples for training and testing of faces.
- CIFAR-10 [26]: 50,000 examples for training and 10,000 for testing with 10 different classes.
- Caltech-UCSD Birds [27]: 6,033 images for training and testing with 200 bird species.
- Oxford-102 Flowers [28]: 8,189 images of 102 flower categories for training and testing.

For this work, 220 simulated images (black and white) of size 576x576px were provided but this number definitively seems too small to perform the training compared to the examples above so image augmentation was performed. Also note that the input examples have no class as this is an unlabelled dataset where no classification task is going to be performed (aside from the discrimination of fake vs real samples). This resulted in a dataset ten times bigger with 2,200 examples. For each image in the original dataset, 2 mirror operations and 7 rotation operations where applied: horizontal mirror, vertical mirror, 45º, 90º, 135º, 180º, 225º, 270º, 315º. Images where also center cropped to a size of 512x512px and downsampled to 256x256px, 128x128px, 64x64px, 32x32px, 16x16px, 8x8px and 4x4px for testing and training purposes. The image ranges where also normalized inside $[-1, +1]$. An example of one of the base images and some of the augmented images are shown in figure 6 along with some of the representations of the downsampled maps.

In order for a faster training to take place, the 128x128px size has been chosen as the desired output size, which as seen in figure 6 is big enough to maintain the general structure and a good quality for a tiling process to take place later.

Moreover, in figure 7 the average values of the sum of all the images in the dataset can be seen, this asserts that there is no underlying common structure

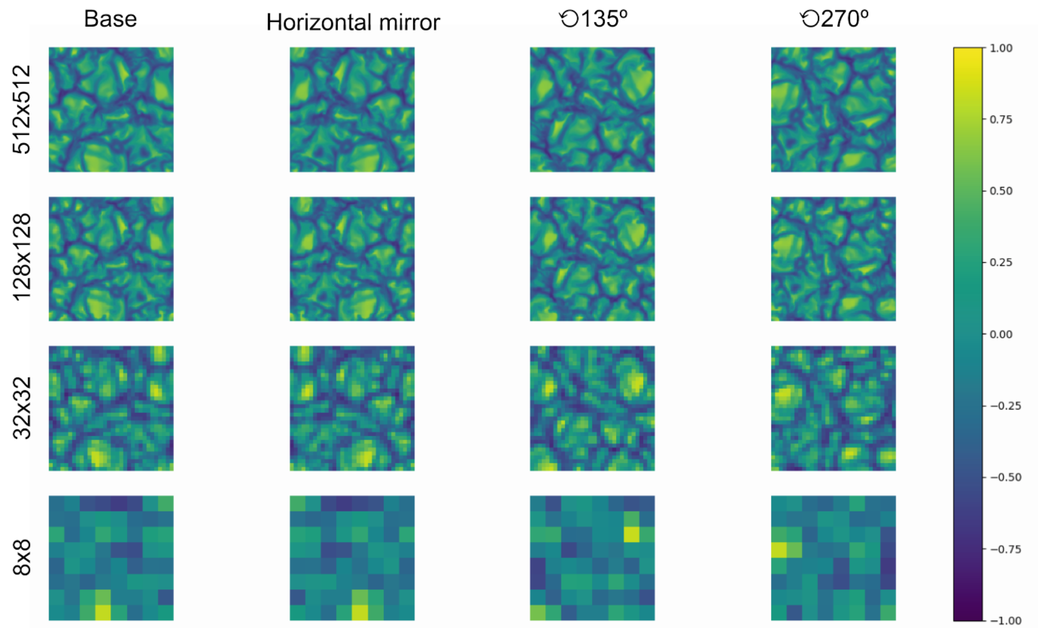


Figure 6: One sample of the solar images dataset with some of the augmenting operations and downsamples.

in all images that the generator may learn to trick the discriminator reaching a point where all new generated samples may seem way too similar or where they are simply sharing a common element in some region. Note how some slightly structure can seen in both subfigures but not enough to cause any problem with the learning process, besides it's clear how the augmented images heatmap is more uniform.

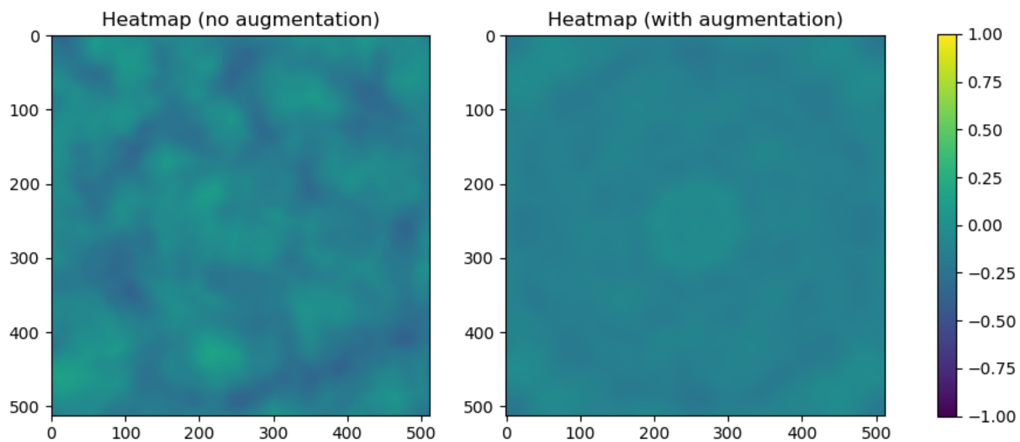


Figure 7: Average values of the sum of all the images in the dataset.

3.2 Generating synthetic solar images with VAEs

Some tests where performed with a VAE, the images shown in figure 8 represent the output of one of several tests. As it was expected, VAEs do not generate images sharp enough for the task at hand and instead we obtain a blurry approximation of the input image. Generating new images (most-right subfigure) from a random latent vector not only had the same results, but also produced a more "square like" pattern. The structure of the encoder and decoder for that specific experiment is defined in table 3 with a total of 8,691,073 trainable parameters. The optimizer was Adam (adaptive moment estimation) with a learning rate of 0.001 and β_1=0.5, $\beta 2$=0.999 along with an MSE loss which is good to represent differences between reconstructions of the image and a Kullback-Leibler divergence loss for the normal distribution.

VAE (hidden dim = 512, latent dim = 256)			
Encoder		**Decoder**	
Output size	**Layer**	**Output size**	**Layer**
1x128x128	Input	512x1x1	UnFlatten input
256x128x128	Conv(k=1, s=1, p=0)	256x4x4	Conv(k=4, s=1, p=3)
256x128x128	LeakyReLU(α=0.2)	256x4x4	LeakyReLU(alphα=0.2)
		256x4x4	BatchNorm
256x64x64	Downsample	256x8x8	Upsample
256x64x64	Conv(k=3, s=1, p=1)	256x8x8	Conv(k=3, s=1, p=1)
256x64x64	LeakyReLU(α=0.2)	256x8x8	LeakyReLU(α=0.2)
256x64x64	BatchNorm	256x8x8	BatchNorm
256x32x32	Downsample	256x16x16	Upsample
256x32x32	Conv(k=3, s=1, p=1)	256x16x16	Conv(k=3, s=1, p=1)
256x32x32	LeakyReLU(α=0.2)	256x16x16	LeakyReLU(α=0.2)
256x32x32	BatchNorm	256x16x16	BatchNorm
256x32x32	Downsample	256x32x32	Upsample
256x16x16	Conv(k=3, s=1, p=1)	256x32x32	Conv(k=3, s=1, p=1)
256x16x16	LeakyReLU(α=0.2)	256x32x32	LeakyReLU(α=0.2)
256x16x16	BatchNorm	256x32x32	BatchNorm
256x32x32	Downsample	256x64x64	Upsample
256x8x8	Conv(k=3, s=1, p=1)	256x64x64	Conv(k=3, s=1, p=1)
256x8x8	LeakyReLU(α=0.2)	256x64x64	LeakyReLU(α=0.2)
256x8x8	BatchNorm	256x64x64	BatchNorm
256x32x32	Downsample	256x128x128	Upsample
32x4x4	Conv(k=3, s=1, p=1)	256x128x128	Conv(k=3, s=1, p=1)
32x4x4	LeakyReLU(α=0.2)	256x128x128	LeakyReLU(α=0.2)
32x4x4	BatchNorm	256x128x128	BatchNorm
512	Flatten	1x128x128	Conv(k=1, s=1, p=0)

Table 3: Detailed architecture of the VAE (k=kernel size, s=stride, p=padding).

The test shown in figure 4 represents the result after 2000 epochs of training which took several hours without any notable improvement in the last 1500

epochs. Similarly to the GAN tests, upsampling layers with convolutions were chosen instead of the equivalent convolution operations alone for upsampling and downsampling using nearest and bilinear interpolation respectively.

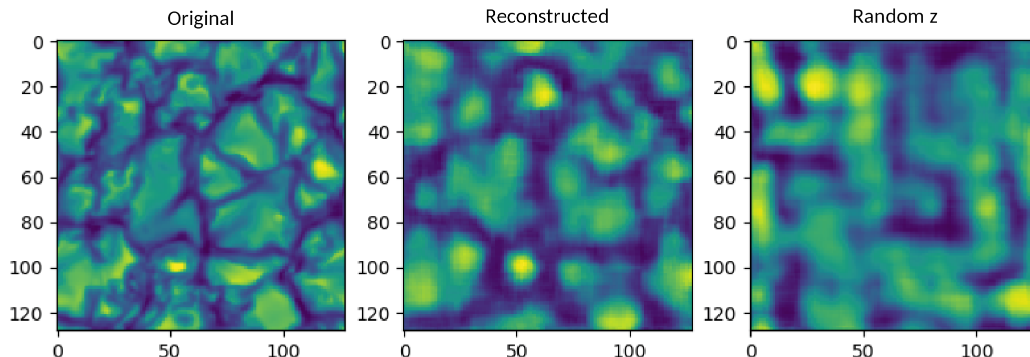


Figure 8: VAE output after 2000 epochs.

3.3 Generating synthetic solar images with GANs

After several experiments, the best network architecture for the GANs is the one proposed in figure 9 and explained in detail in table 4 with a total of 12,848,641 trainable parameters for the generator and 11,813,377 for the discriminator. Generator and discriminator have intentionally mirroring structures, and in both the usage of 512 as the depth of the features maps is an idea obtained from some of the PGGAN implementations [3] where the usage of a smaller number of layers is not required until the output resolution is big enough, more than the 128x128px target of our network.

Even though, progressive growing was tested for this matters it seemed to produce worse or slower results in this use case than directly training with the final resolution. The combination of both networks with the final architecture was trained for a total of 7 hours and 120 epochs which is an appropriate amount of time given that the final representation seem to have a good complexity. Note how according to the table 4 the input to the generator is a latent vector of size 128. The size of this randomly sampled vector seems to fit perfectly the problem, with bigger input sizes (512) the samples generated by the network where very similar or of small variability.

The exercise of finding the best architecture for this methodology has been an actual fine tuning process where different approaches from the state of the art knowledge has been tested. The batch size for this problem is 8, it couldn't be bigger due to memory restrictions with images of size 128x128px but this is not an issue since a small batch size is the recommendation according to the general consensus around GANs. Similarly to the VAE, the optimizer was Adam with a learning rate of 0.001, and β_1=0, $\beta 2$=0.99. In this case, following

[3]https://github.com/nashory/pggan-pytorch

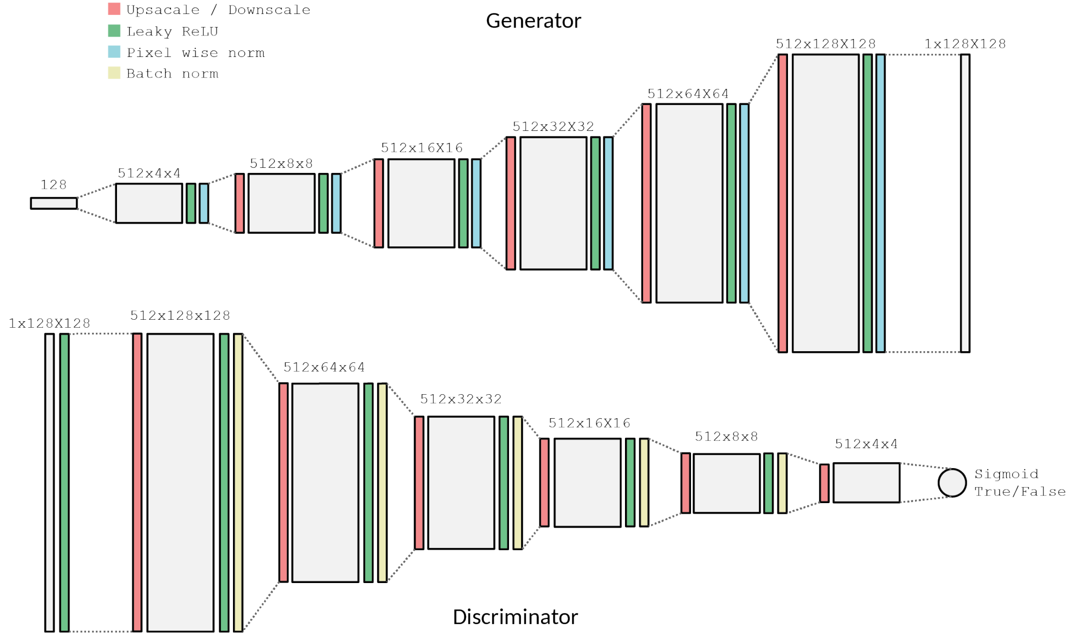
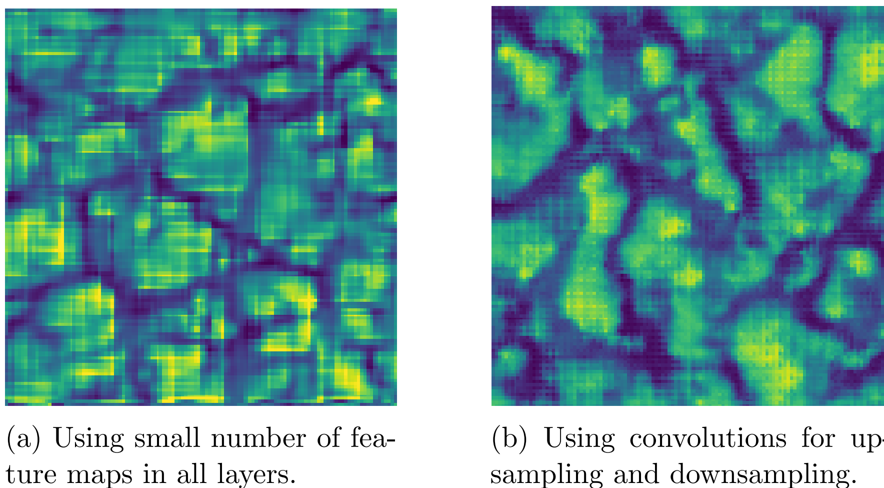


Figure 9: GAN representation (red=scaling, green=leaky ReLU, blue=pixel normalization, yellow=batch normalization).

some of the already existing implementations, the β_1 parameter was directly set to 0, the decaying rate for the gradient exponential moving average which in turn gave better results. Binary crossentropy is chosen as the loss function for the discriminator being this error is propagated to the generator as well. For this problem, decreasing the number of feature maps from 512 into 256 produced a huge number of artifact: figure 10a and using only convolutional layers instead of upsampling (nearest neighbour) and downsampling (bilinear) resulted in noticeable checkerboard artifacts.



(a) Using small number of feature maps in all layers.

(b) Using convolutions for upsampling and downsampling.

Figure 10: Artifacts generated by different techniques (size 128x128px).

For the convolution layers, in the GAN and VAE networks, equalized learning was used with the Kaiming Normal [29] for the weight normalization that pre-

GAN			
Generator		**Discriminator**	
Output size	**Layer**	**Output size**	**Layer**
128x1x1	Input	1x128x128	Input
512x4x4	Conv(k=4, s=1, p=3)	512x128x128	Conv(k=1, s=1, p=0)
512x4x4	LeakyReLU(α=0.2)	512x128x128	LeakyReLU(alphα=0.2)
512x4x4	PixelWiseNorm		
512x8x8	Upsample	512x128x128	Conv(k=3, s=1, p=1)
512x8x8	Conv(k=3, s=1, p=1)	512x128x128	LeakyReLU(α=0.2)
512x8x8	LeakyReLU(α=0.2)	512x128x128	BatchNorm
512x8x8	PixelWiseNorm	512x64x64	Downsample
512x16x16	Upsample	512x64x64	Conv(k=3, s=1, p=1)
512x16x16	Conv(k=3, s=1, p=1)	512x64x64	LeakyReLU(α=0.2)
512x16x16	LeakyReLU(α=0.2)	512x64x64	BatchNorm
512x16x16	PixelWiseNorm	512x32x32	Downsample
512x32x32	Upsample	512x32x32	Conv(k=3, s=1, p=1)
512x32x32	Conv(k=3, s=1, p=1)	512x32x32	LeakyReLU(α=0.2)
512x32x32	LeakyReLU(α=0.2)	512x32x32	BatchNorm
512x32x32	PixelWiseNorm	512x16x16	Downsample
512x64x64	Upsample	512x32x32	Conv(k=3, s=1, p=1)
512x64x64	Conv(k=3, s=1, p=1)	512x32x32	LeakyReLU(α=0.2)
512x64x64	LeakyReLU(α=0.2)	512x32x32	BatchNorm
512x64x64	PixelWiseNorm	512x8x8	Downsample
512x128x128	Upsample	512x8x8	Conv(k=3, s=1, p=1)
512x128x128	Conv(k=3, s=1, p=1)	512x8x8	LeakyReLU(α=0.2)
512x128x128	LeakyReLU(α=0.2)	512x8x8	BatchNorm
512x128x128	PixelWiseNorm	512x4x4	Downsample
1x128x128	Conv(k=1, s=1, p=0)	1x1x1	Conv(k=4, s=1, p=0)
		1	Sigmoid

Table 4: Detailed architecture of the GAN (k=kernel size, s=stride, p=padding).

vents the situation where some paramaters take longer to adjust than others if they have a larger dynamic range, this scenario may be caused by modern initializers where the learning rate can be at the same time too big and too small for different parameters [16]. The use of equalized learning did improve the generation of new samples as well as changing the batch normalization in the generator for a pixel wise normalization which also prevents the escalation of signal magnitudes.

In order to assert that no repeating patterns or common structures have been learned for all output samples, heatmaps are also generated for the first epochs as an example, see figure 11, it can be seen how they are not so uniform as the inputs, but seem to be randomly distributed and good enough to be used as a substitute of the real images.

In figure 12, some samples generated by the GAN at different epochs is shown.

It's often observed in these networks how not necessarily more epochs indicate a better output sample by the generator, remember how this is a game where the balancing between discriminator and generator is not always ideal. Epochs 60, and 20 seem to throw the best outputs according to a human interpreter (zoomed examples of epoch 60 in figure 13), so the recommended network to use is the one saved at epoch 20, as it is closer to an uniform distribution as shown in the previous heatmaps.

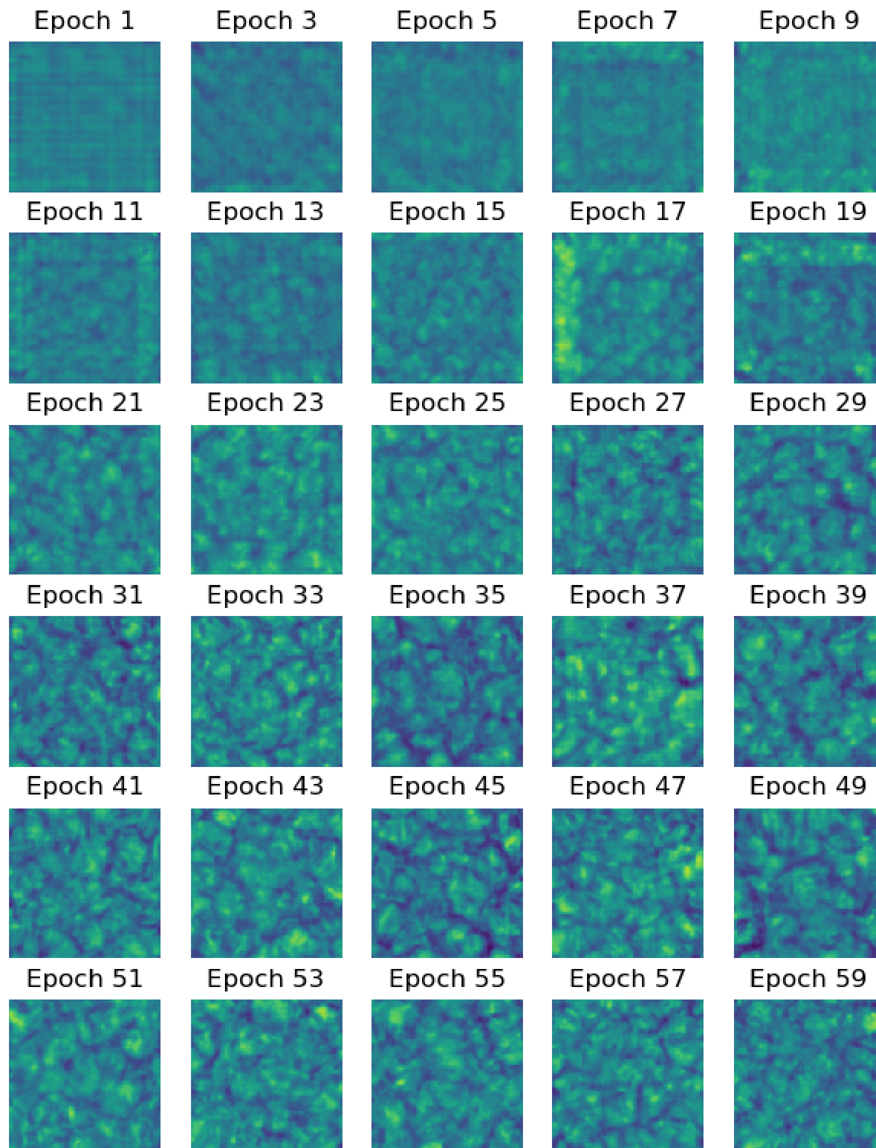


Figure 11: Average values of the sum of 100 generated images in each epoch.

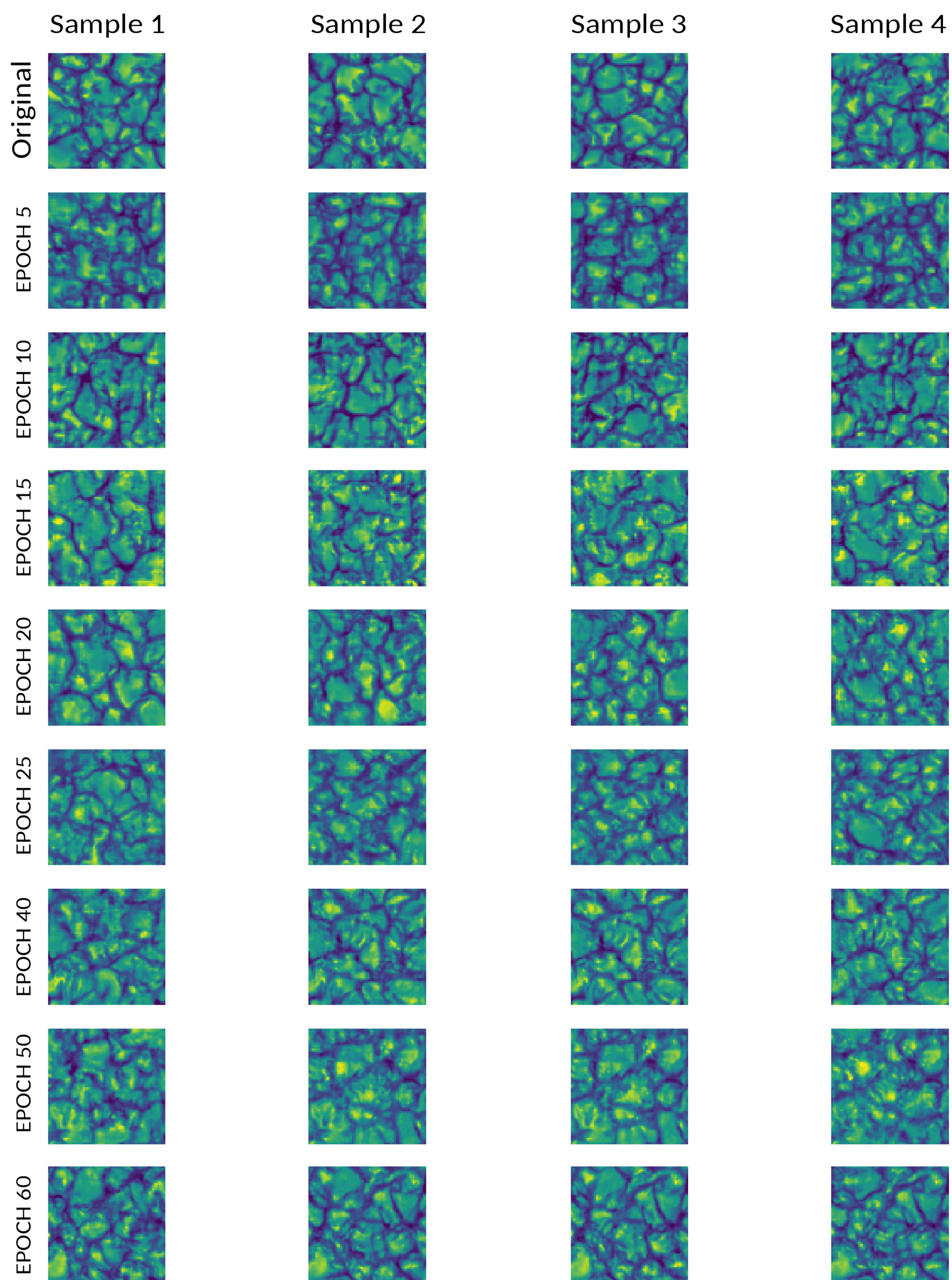


Figure 12: GAN outputs at different epochs.

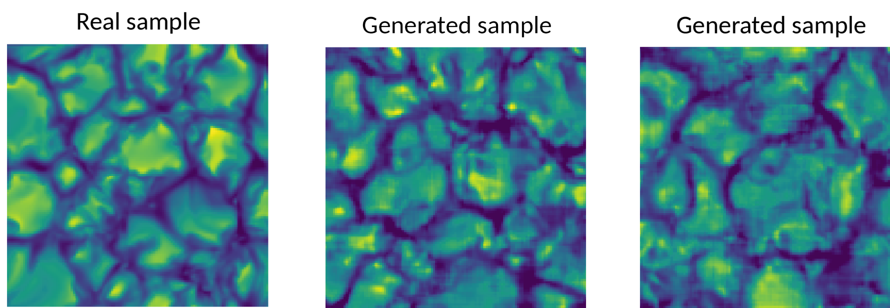


Figure 13: Zoomed example of generated samples at epoch 60.

3.4 Tile stitching

Now that the generator has been trained the remaining steps include the generation of a composite image of any given size. For this, the generator is asked to produce a huge number of images that will then be stitched together. Note that it's impossible for the stitching process to be faster than $O(nm)$ in a single core CPU (or a single process program) where n and m are the output required size in the number of tiles if no repeating tile is expected. For the same reason, memory consumption will be expected to be at least of size $O(nm)$, however, since a big enough image will eventually not be able to fit in the graphics memory (VRAM) and for a good quality in the composition, more than $m*n$ images are required (to find the best subset of images that form the highest quality composited image), the generation of images is processed in batches of small size (approximately 10 to 25 images) depending on the available VRAM if any.

It's not recommended, however this process may also take place exclusively in the CPU with the consequent time increment, at least by a factor of 10. The time taken to generate each batch of candidates tile is shown in table 5, see how for small batch sizes that fit in VRAM, time variations are negligible. Thus the time taken depends on how many batches are needed and roughly not by the size of those batches (unless there's a big difference in batch size).

For continuous usage, the generator could be left alone generating a huge amount of images (e.g. 1,000,000) just once and each time an image of any size (smaller than $m = n = \sqrt{1,000,000}$) shall be generated, a random subset of size $10 * m * n$ is chosen from the backup file.

Batch Size	**Time GAN (s)** $\bar{t}$	**Time GAN (s)** σ	**Time VAE (s)** $\bar{t}$	**Time VAE (s)** σ
20	$2.99 * 10^{-3}$	$1.48 * 10^{-5}$	$2.68 * 10^{-1}$	$1.03 * 10^{-1}$
15	$3.20 * 10^{-3}$	$4.38 * 10^{-4}$	$1.45 * 10^{-1}$	$9.79 * 10^{-2}$
10	$3.58 * 10^{-3}$	$5.48 * 10^{-4}$	$1.39 * 10^{-2}$	$1.09 * 10^{-2}$
5	$3.40 * 10^{-3}$	$8.77 * 10^{-4}$	$1.93 * 10^{-2}$	$2.47 * 10^{-2}$
1	$3.39 * 10^{-3}$	$5.45 * 10^{-4}$	$7.91 * 10^{-2}$	$8.30 * 10^{-2}$

Table 5: Time taken for a forward pass.

3.4.1 Blending similar tiles with a greedy search

A first approach for the building of a composite image is a greedy search. N ($N >= m * n$) number of tiles will be generated and the best matching tile for each row and column will be selected each time while constructing the final image. The building process goes row first from left to right and then columns top to bottom, the first tile is chosen randomly. Matching tiles are selected based when an overlap region is defined: amount of the image boundaries that will be blended with their neighbours. For a tile size of 128x128px, good overlap region sizes are 10, 15 or 20.

When tile must be find that matches with a tile at its left, distances are measured as the difference between the most right pixels (overlapping region) for the left tile and the most left pixels (overlapping region) for the right candidate tile, equation 3. The tile from the remaining N pool of initial tiles with the smallest difference is chosen, thus completing a greedy search.

$$d(left, right) = \sum_{x=0}^{overlap-1} \sum_{y} left(x - overlap, y) + right(x, y) \tag{3}$$

Tiles are then blended together, this blending may occur in the latent space (see next section) or in the generated tile image. If the blending is performed in the final image, the pixels in the overlapping region will be computed according to equation 4 (where negative pixels indicate that the pixel count starts from the right side). The value of α changes for each x according to the number of columns in the overlapping region starting at $\alpha = 1$.

$$pixel(x, y) = left(x - overlap, y) * \alpha + right(x, y) * (1 - \alpha) \; : \; \alpha \in [0, 1] \tag{4}$$

Since pixels are going to be blended together, a more accurate distance metric could be obtained when the search tries to minimize the distance between the central column of pixels, where $p(middle, y) = left(middle, y) * 0.5 + right(middle, y) * 0.5$, and gives a smaller importance to the difference of pixels in the opposite side: $p(0, y) = left(-overlap, y) * 1 + right(0, y) * 0$ or $p(overlap, y) = left(-1, y) * 1 + right(overlap, y) * 0$. Thus, the distance function in equation 3 is updated to include a gaussian 1D kernel that gives more importance to the middle column (equation 5).

$$d(left, right) = \sum_{x=0}^{overlap-1} K_x \sum_{y} left(x - overlap, y) + right(x, y) \tag{5}$$

The same operation is taken to measure the distance between a top tile and a candidate bottom tile. And lastly, when the candidate tile (right-bottom) has to compare itself with a left tile and a top tile, the maximum of the two distances is taken as the distance for that tile.

However the question arises of how many tiles should the generator produce for a good quality output image following this method. As an experimental approach images of different target sizes in number of tiles with different initial pool sizes of generated tiles where produced and the average distance of all the selected tiles is calculated in figure 15.

Note that to have a substantial difference the number of samples must be logarithmically increased, however, for a good average distance, a pool size of an order of magnitude greater than the size of the image in tiles is recommended.

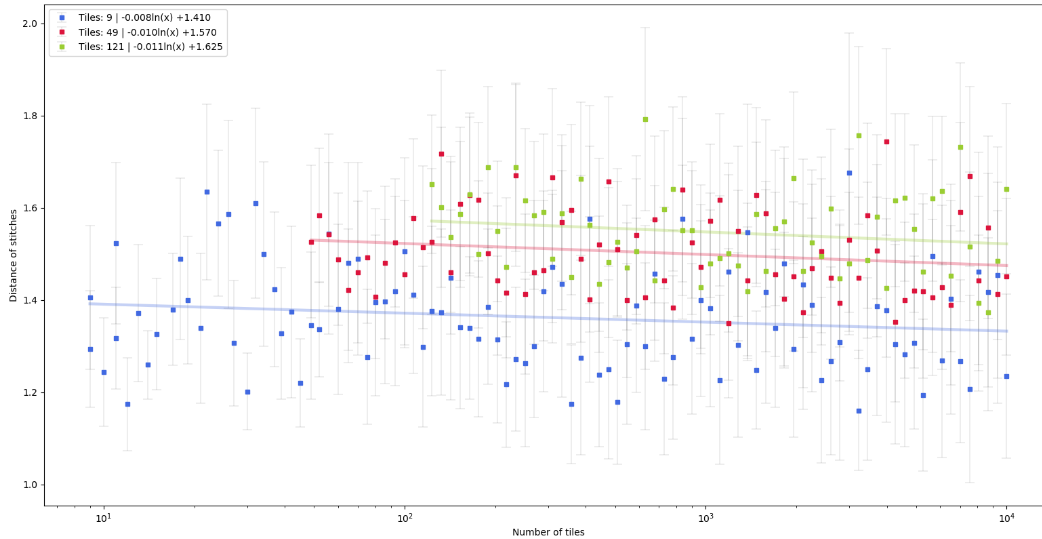


Figure 14: Average distance of tile matching.

3.4.2 Blending similar tiles with a greedy search in latent space

The same process of blending depicted above can be carried out in the latent space of the generator, because the convolutions and upsampling layers of the network are not transform or location invariant, changes implemented in feature maps of a lower latent space will be translated into the final image without changing their location (i.e. they will stay in the overlapping region of the tile image) thus in theory resulting in a better image to perform the blending operation on.

Images are again chosen using the same distance metric as before, but the blending takes place between latent spaces of both images and then, between the final modified images. Note that the overlapping region size is different in the latent space of the feature maps.

However, for this problem, the improvement obtained from blending the feature maps from an intermediate layer are almost imperceptible, not only for the human eye, but the changes at the pixel level are insubstantial compared to the simpler approach of just blending the output images.

3.4.3 Refining tiles with a GA

Another approach where the resulted composited image is expected to improve in quality is a method that uses Genetic Algorithms (GA) for the modification of the latent input vector of the generator.

The first steps are the same as before, a huge number of images (at least $10 * m * n$) are generated or loaded from file and for each tile a best match is searched using the distance function against all other tiles. However, now that the tile has been chosen, a GA takes place to try and improve the vector z of the current tile searching for the value that lies in the local minima of

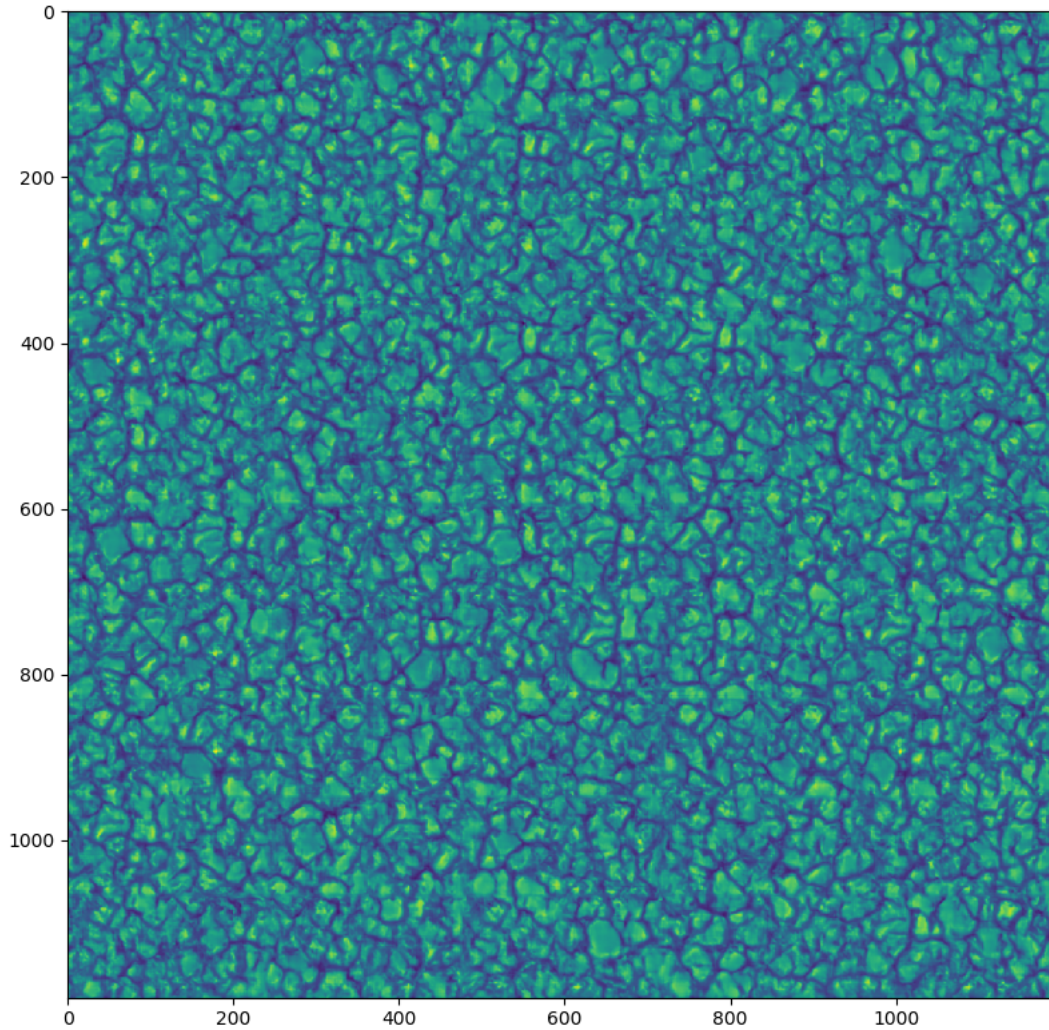


Figure 15: Texture resulted from stitching and blending together 100 tiles in a 10x10 square with an overlapping region of $10px$. Composited image size is $1190 \text{x} 1190px$.

the fitness function (note that for this, not only we need to store the output images for the generator but also the input vectors that created them):

1. A population of N individuals is created from:
   (a) A copy of the original tile z vector.
   (b) Small variations of the original tile z vector.
   (c) Considerable variations of the original tile z vector.
   (d) Random z vectors.
2. Calculate the fitness of each individual:
   (a) Generate the output image for each z vector.
   (b) Calculate the distance from the output image to the image it must match.
   (c) Sort individuals by fitness.

3. Select top N_1 individuals to stay in the next generation (elite).
4. Generate N_2 individuals using cross over from two individuals:
   (a) Choose parent A using roulette selection with the fitness.
   (b) Choose parent B using roulette selection with the fitness.
   (c) Crossover A and B with single point random crossover over their z vector.
5. Generate N_3 individuals completely random (random z vectors).
6. Mutate all z vectors with a random gaussian distribution ($x \in \mathcal{N}$ & $x \in [-1, +1]$) for each item in vector.
7. Go to (2) and repeat, unless an individual with a fitness value above the threshold has been found or the maximum number of iterations reached.

A good value for the population $N = N_1 + N_2 + N_3$ is around 50 or 100, nonetheless the bigger the population the more batches or chunks of images that need to be calculated on the GPU (or CPU) thus slowing down the calculation of the composite image. Experimentally, N_1 and N_3 should be of around 10% of the total population size each one. In figure 16 an example comparing the improvements of this technique are shown.

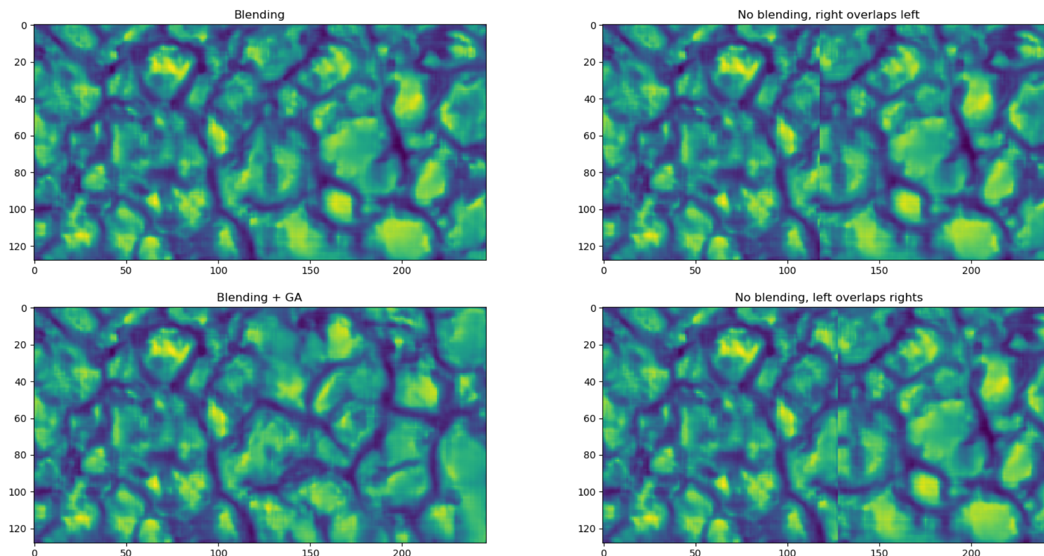


Figure 16: Improvements given by the GA approach.

This technique however slows down the generation of each tile. Incrementing the time needed from selecting a tile from the generated database from less than a second to almost a minute, depending on the fitness threshold and the maximum number of iterations allowed.

4 Software

For this work as a side objective, a desktop application was created to facilitate the generation of the solar surface images to any user. The code can be found in the Github repository: https://github.com/Ediolot/solari. Even though this software comes with said desktop application, it can also be used as a simple python program whether it will be run from the *main.py* file following the instructions in the *readme.md* or be the functions imported to a different python program. A screenshot of the desktop application is found in figure 17.

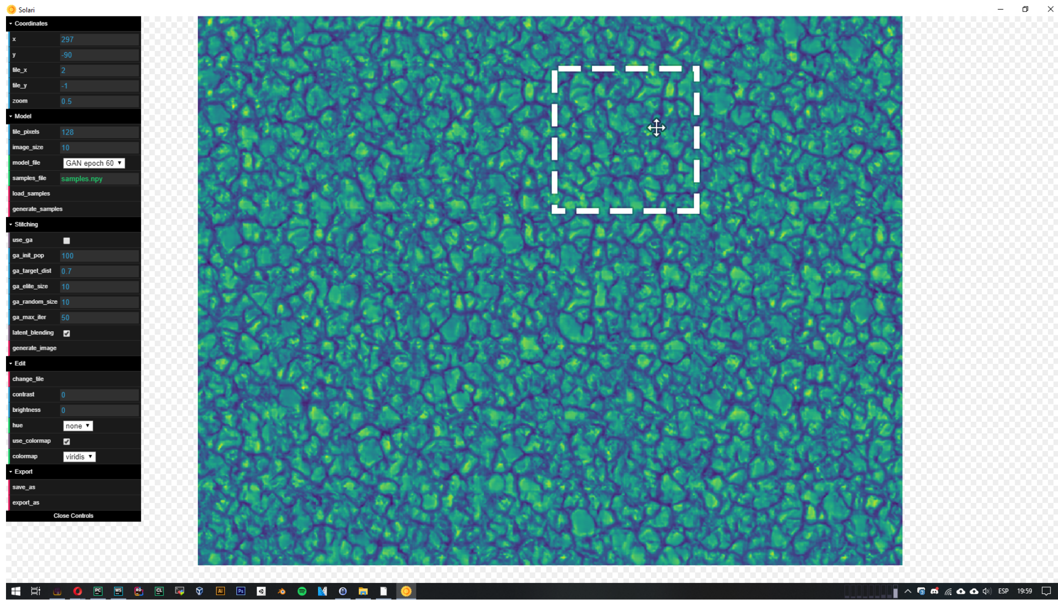


Figure 17: Screenshot of the desktop application

The desktop application comes with the basic panning, rotating and zooming tools as well a complete tool menu with the necessary commands to generate new database samples or load them from file, change the overlapping amount, define the usage of the GA refinement, use the GPU or CPU, exporting the image, etc... (not every option is implemented in the current version but it's an ongoing work).

As explained before, one downside of the GA refinement method is that it slows down the generation of the composite image by approximately an order of magnitude. If that's the case, the user may want to simply generate an image using the basic blending method and then within the software select which tiles have the lowest quality and replace them using the GA technique.

For the desktop application a web based application was developed, because it tends to be more user friendly at the expense of a more complex development and a bigger amount of resources dedicated for the GUI. The interface is connected to the python back-end through a local server that executes the requested commands and sends back the data and the images for the user.

The complete list of technologies used is: NodeJS, Javascript, HTML, CSS, Electron, Python and Flask.

5 Conclusions

The proposed method shows that it's possible to generate the needed images without spending a big amount of time. Compared to the previous method which requires a vast amount of computational power and as such the need to reserve access to the *LaPalma* supercomputer for this task, this new method allows any user to produce images of good quality easily which are indeed not similar to one another contrary to what happens in a continuous simulation.

Some examples of the final images can be seen in the Annex, in figures 18, 19, 20 and 21. The biggest image of size 5910x5910px took only 28 minutes while the smaller images take seconds.

Although the tiling size has been reduced from 128x128px from 512x512px the structure of generated tiles is effortlessly identified and the resolution error is unnoticeable for big images. The most simple tiling method have been shown to be effective, as well as the improvement proposed of the GA refinement. The latent feature maps blending didn't show any real improvement, but it may be required for bigger output sizes in the generator.

As it was expected the VAE didn't perform well enough for this task generating only blurred images with no real application inside the requirements. An adversarial network with a similar number of layers and feature maps performed many times better than the VAE producing sharper images. This task in particular needs sharp images, this does not come as a surprise, as seen in the original dataset, images are formed from a number of blobs that tend to have an an uniform interior and are strongly limited at their boundary by a sharp change in the image values.

Despite having access to a variety of powerful resources to complete the training processes, the fine tuning and testing of different layers and parameters for the four networks (GAN: Generator, GAN: Discriminator, VAE: Encoder, VAE: Decoder) took a good amount of time that spans around the duration of several months. Training these networks take a fair amount of time and the smallest change effectively needs a new training which slows down all the process.

Finally the development of a desktop tool to fulfill the needs of any user is a good addition to the software, this tool has proven to work flawless in most cases.

6 Future work

Two important points remain as possible future work, first, image tiles shall be generated with the correct size of 512x512px, for this task, two solutions are immediately available:

1. An upsampling CNN that transforms tiles from the generator into the correct size.

2. Expand the generator to output the correct size, which in turns also needs to expand the discriminator layers at the input region.

Multiple models of upsampling networks exist (e.g. this software [4] augments the size of photos and cartoony images). Images may be resized using bilinear, bicubic or any other interpolation technique into the desired size and then they are processed through a CNN that does not modify the size of the image. The CNN detects patterns and structures in the original image and outputs a sharper and denoised image. The biggest difference is that those are general purpose models, and for this problem only a specific model for this task is needed.

Another point for improvement lies in the tiling, as seen before, some previous work [19] explain how optimal seams are found for the generation of new textures from a previous sample. This process could be used as well for the stitching of the different generated tiles into the final composite image. A further improvement involves more advanced search techniques for the GA which now is just configured as a random search algorithm, but it could be modified in order for it to generate mutations that are more likely to lay closer to the local minima.

[4] https://github.com/nagadomi/waifu2x

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[2] Alexei Efros and Thomas Leung. Texture synthesis by non-parametric sampling. In *In International Conference on Computer Vision*, pages 1033–1038, 1999.

[3] W.T. Freeman, T.R. Jones, and E.C. Pasztor. Example-based super-resolution. *Computer Graphics and Applications, IEEE*, 22:56–65, 04 2002.

[4] James Hays and Alexei Efros. Scene completion using millions of photographs. *ACM Trans. Graph.*, 26:4, 07 2007.

[5] Yoshua Bengio, Li Yao, Guillaume Alain, and Pascal Vincent. Generalized denoising auto-encoders as generative models, 2013.

[6] Shengjia Zhao, Jiaming Song, and Stefano Ermon. Towards deeper understanding of variational autoencoding models. 02 2017.

[7] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric. *CoRR*, abs/1512.09300, 2015.

[8] Michael J. Smith and James E. Geach. Generative deep fields: arbitrarily sized, random synthetic astronomical images through deep learning. 2019.

[9] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.

[10] Andrew L. Maas. Rectifier nonlinearities improve neural network acoustic models. 2013.

[11] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *ArXiv*, abs/1505.00853, 2015.

[12] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2015.

[13] Moritz Hardt, Benjamin Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *ICML*, 2015.

[14] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014.

[15] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 2016.

[16] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *ArXiv*, abs/1710.10196, 2017.

[17] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans, 2016.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[19] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: Image and video synthesis using graph cuts. *ACM Trans. Graph.*, 22:277–286, 07 2003.

[20] Rohan Chandra, Sachin Grover, Kyungjun Lee, Moustafa Meshry, and Ahmed Taha. Texture synthesis with recurrent variational auto-encoder. *ArXiv*, abs/1712.08838, 2017.

[21] Nikolay Jetchev, Urs Bergmann, and Roland Vollgraf. Texture synthesis with spatial generative adversarial networks. *ArXiv*, abs/1611.08207, 2016.

[22] Urs Bergmann, Nikolay Jetchev, and Roland Vollgraf. Learning texture manifolds with the periodic spatial gan. In *ICML*, 2017.

[23] Anna Frühstück, Ibraheem Alhashim, and Peter Wonka. Tilegan: synthesis of large-scale non-homogeneous textures. *ACM Trans. Graph.*, 38:58:1–58:11, 2019.

[24] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[25] A. K. Anderson J. M. Susskind and G. E. Hinton. The toronto face database. Technical Report UTML TR 2010-00, University of Toronto, 2010.

[26] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[27] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie. The Caltech-UCSD Birds-200-2011 Dataset. Technical Report CNS-TR-2011-001, California Institute of Technology, 2011.

[28] M-E. Nilsback and 2008 Zisserman, A. Oxford 102 flowers.

[29] Zhaodong Chen, Lei Deng, Bangyan Wang, Guoqi Li, and Yuan Xie. A comprehensive and modularized statistical framework for gradient norm equality in deep neural networks, 01 2020.

7 Annex

7.1 Example of some generated images

Some of the generated images are shown here. All the images where generated using the same 10,000 generated dataset constructed from the generator.

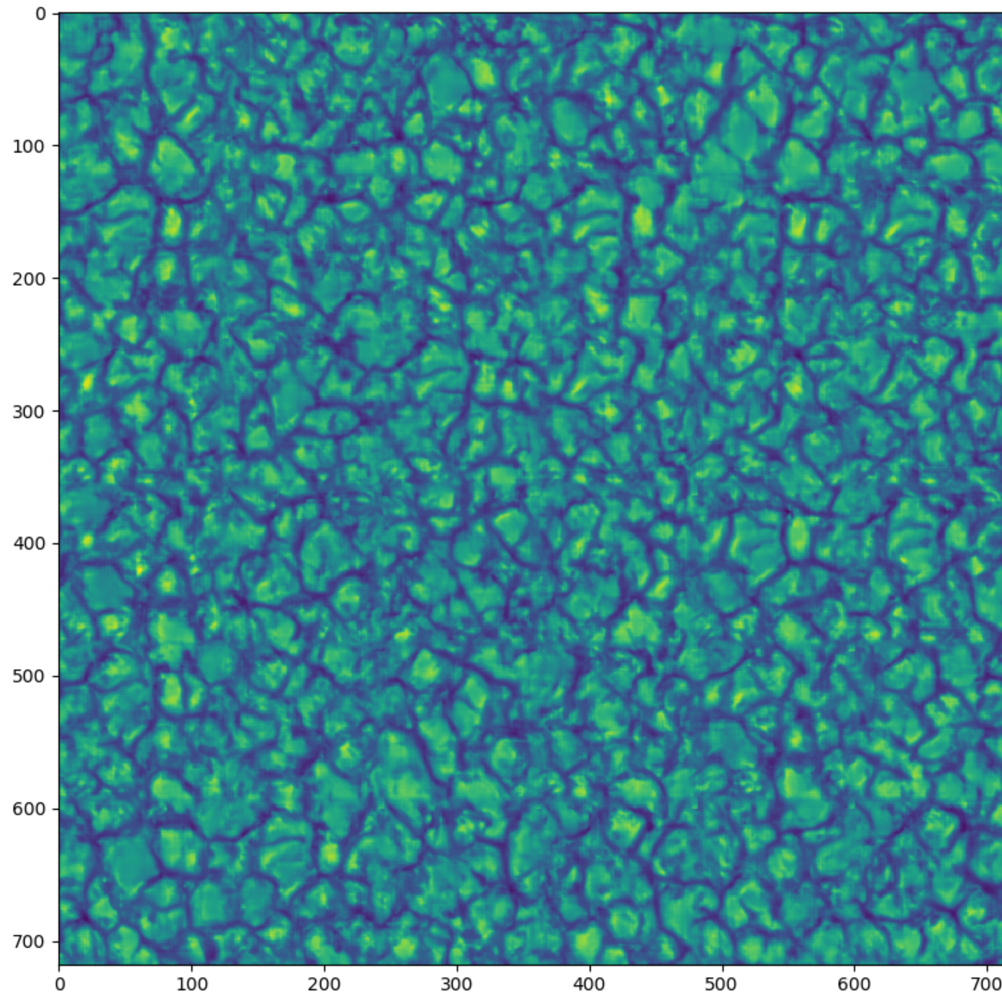


Figure 18: 6x6 tiles, 718x718px (overlap=10px), took 2.16 seconds.

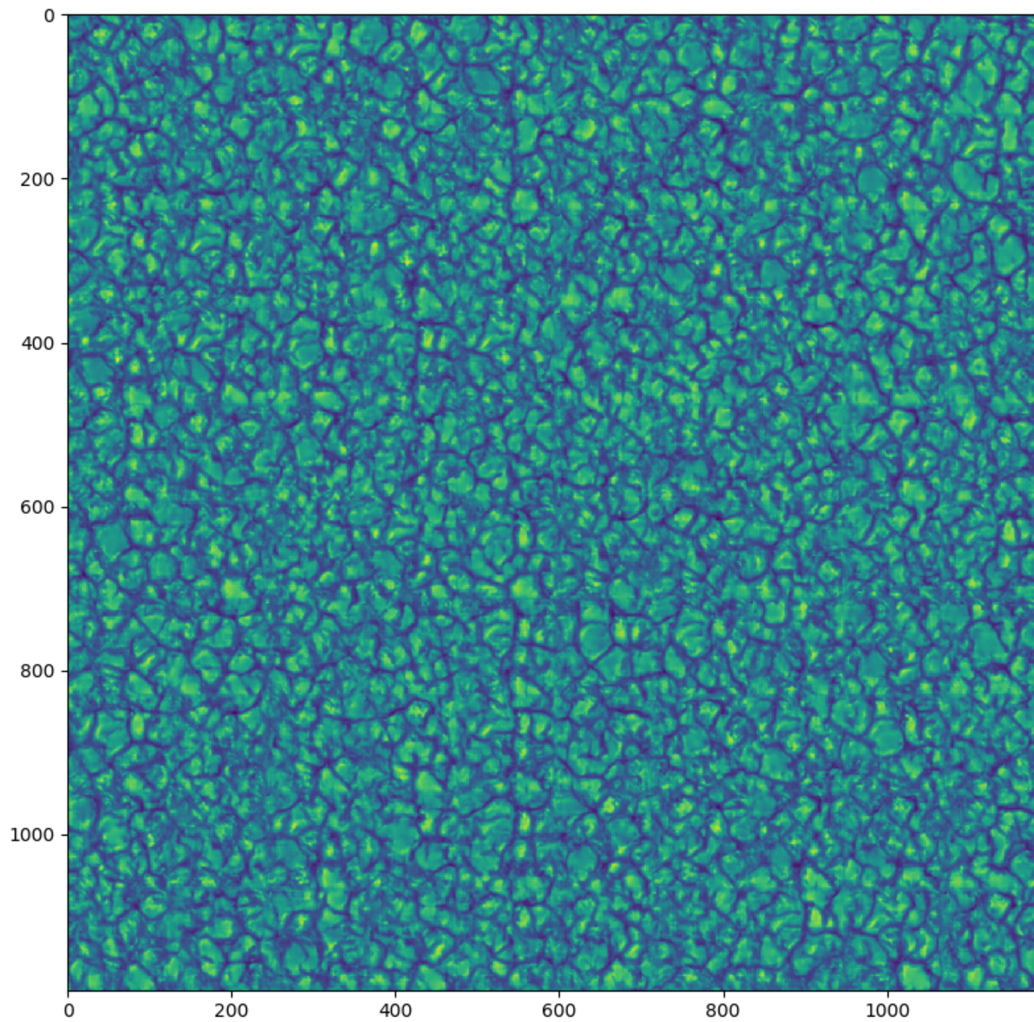


Figure 19: 10x10 tiles, 1190x1190px (overlap=10px), took 6.50 seconds.

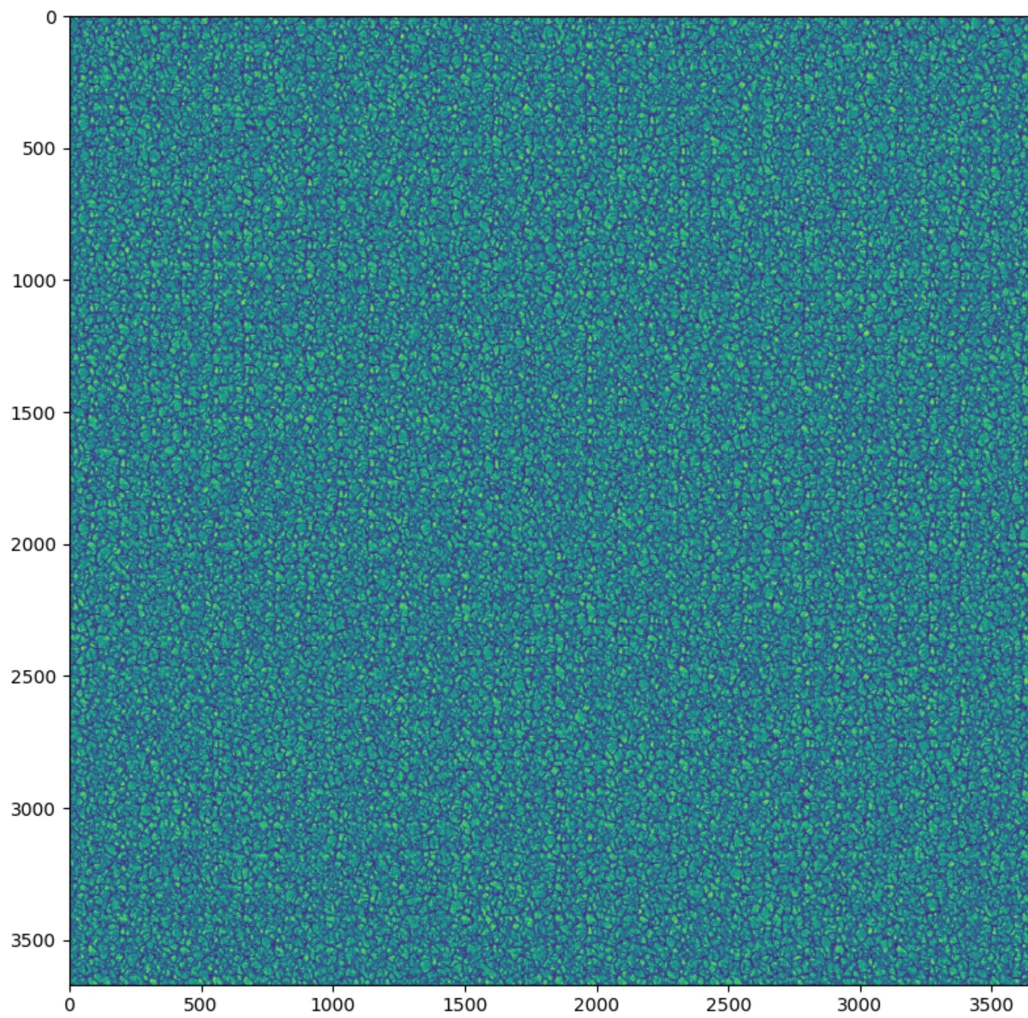


Figure 20: 31x31 tiles, 3668x3668px (overlap=10px), took 11 minutes and 37.02 seconds.

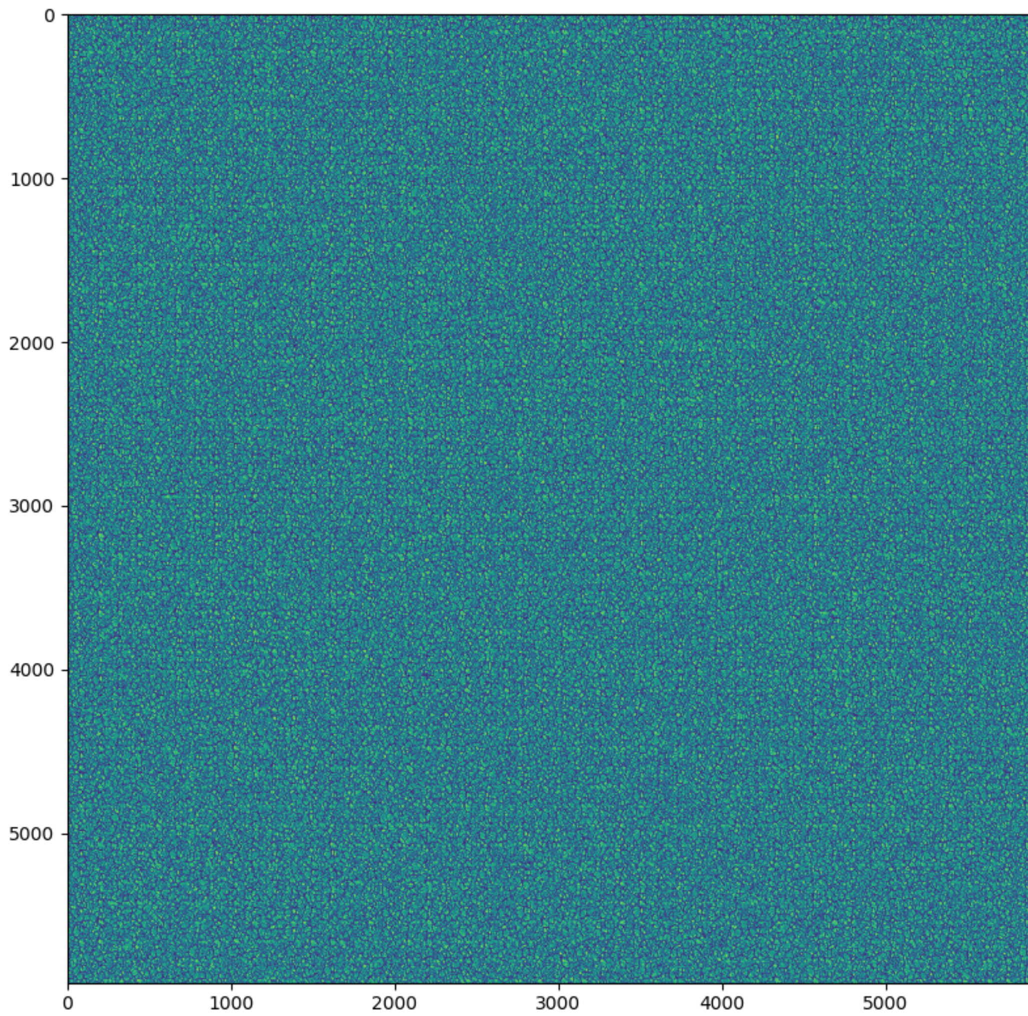


Figure 21: 50x50 tiles, 5910x5910px (overlap=10px), took 28 minutes and 28.58 seconds.