Image Generation with Generative Adversarial Networks from a Single Image (SinGAN)

An analysis of SinGAN's generation capabilities

Sami AMRANI, Hamdi BEL HADJ HASSINE

February 12, 2022

Abstract

In this project we explore one of the recent state-of-the-art image generation techniques which won the ICCV 2019 Best Paper Award. This method, named SinGAN, is based on Generative Adversarial Networks (GANs) and has the specificity of only requiring one single image for training. Our goal in this project is to analyze the images generated by SinGAN from an originality point of view, i.e. to evaluate whether the images generated by SinGAN are completely synthetic images or if they contain patches copied from the original image. To do this we implemented a patch copy-move detection algorithm using PatchMatch and we used it to detect patch copies in the images generated by SinGAN.

1 Introduction

Image generation is a booming area of research. It knew a great improvement after the advent of GANs, designed by Y. Goodfellow et al [1], and of Variational Autoencoders, introduced by Kingma et al [2]. Both of these methods are Deep Learning algorithms, and were born amid the tide of its development, relying on the accessibility of massive amounts of data and better than ever GPUs.

GANs, in particular, are described by Yann LeCun as "the most interesting idea in the last 10 years in Machine Learning". It consists in two neural networks with a symmetric architecture : the Generator, and the Discriminator. The Generator is trained to fool the Discriminator by generating data with a distribution as close as possible to that of real data. The Discriminator is trained to discriminate between the fake outputs of the Generator, and real data.

During the last 7 years, GANs have been adapted to tackle a great range of problems, like super-resolution [3], inpainting [4], and retargeting [5]. However, despite great success, the main drawback of these models is their need for a massive amount training data, and their lack of generalization.

In 2019, SinGAN was proposed by Shaham et al [6] to tackle this problem. It's a powerful generative model based on GANs that learns the internal statistics of patches within a single natural image. It has the specificity to train on a single image, and can be used for a wide range of image manipulation tasks : image generation, editing, super-resolution, paint-to-image, harmonization, and animation. It consists in a pyramid of fully convolutional

light-weight GANs, each responsible for capturing the distribution of patches of the image at a different scale.

In [6], they rely mainly on Amazon Mechanical Turk (AMT) "Real/Fake" user study and a single-image version of the Fréchet Inception Distance [7] to evaluate the performance of SinGAN. The "Real/Fake" user study is costly and subjective, and Fréchet Incepton Distance is hard to interpret. Both of these measures do not convey much information about the images generated. Since SinGAN is a polyvalent image generation algorithm, we need a better way to analyse and quantify the originality of the images it generates.

Since SinGAN captures the internal distribution of patches in a single image, it seems more intuitive to focus on the similarity of patches, instead of using more general image similarity metrics.

In this project we focus on the application of classical image manipulation tools to measure the originality of SinGAN. Namely we'll use a Copy-Move forgery detection algorithm, developed by Cozzolino et al [8], to detect the patches of the training image that appear in generated images.

This algorithm relies on PatchMatch [9], an algorithm that allows for the quick analysis of the similarity between the patches of two different images. Not the first algorithm to do so, it drastically improved the speed of the computation of nearest neighbors mappings, allowing for its use in many different domains and being at the core of many algorithms.

The proportion of copied patches in turn gives us an indication of SinGAN's originality, allowing us to assess the quality of the generated images.

2 SinGAN: Architecture and capabilities

2.1 How SinGAN works

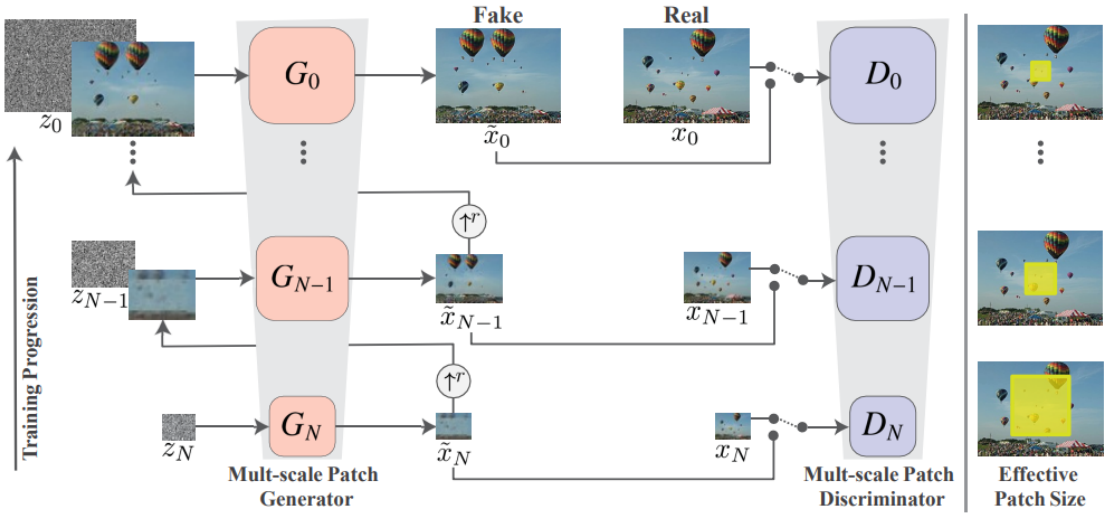


Figure 1: SinGAN architecture. Image credit: [6]

SinGAN consists of a pyramid of PatchGANs $\{(G_0, D_0), ..., (G_N, D_N)\}$, trained on an image pyramid of $x = x_0, ..., x_N$, where x_n is a downsampled version of x by a factor r^n, for some $r > 1$.

PatchGANs [10] consist in a fully convoluted Generator and Discriminator. The Discriminator uses its convolutional layers to classify if each patch in an image is real or fake, and then averages all responses to provide its final output.

The generation of an image sample starts at the coarsest scale and sequentially passes through all generators up to the finest scale, with noise injected at every scale. All the generators and discriminators have the same receptive field and thus capture structures of decreasing size as we go up the generation process. In addition to spatial noise z_n, each generator G_n accepts an upsampled version of the image from the coarser scale, except at the coarsest scale, where the generation is purely generative. G_n tries to reconstruct the missing details in $(\tilde{x}_{n+1}) \uparrow^r$ with its convolutional layers, that is

$$\begin{aligned}\tilde{x}_N &= G_N(z_N) \\ \tilde{x}_n &= G_n(z_n, (\tilde{x}_{n+1}) \uparrow^r), \ n < N \\ &= (\tilde{x}_{n+1}) \uparrow^r + \phi_n(z_n + (\tilde{x}_{n+1}) \uparrow^r)\end{aligned}$$

where ϕ_n are the convolutional layers of the Generator n.

This architecture is trained sequentially, from the coarsest scale to the finest one. Once each GAN is trained, it is kept fixed. The training loss for the n^{th} GAN is comprised of an adversarial term and a reconstruction term :

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

The adversarial loss $\mathcal{L}_{\text{adv}}$ penalizes for the distance between the distribution of patches in x_n and the distribution of patches in generated samples $\tilde{x}_n$. It consists in a WGAN-GP loss [11].

The reconstruction loss $\mathcal{L}_{\text{rec}}$ insures the existence of a specific set of noise maps that can produce x_n. Mainly, during training a $\{z_N^{\text{rec}}, \dots, z_0^{\text{rec}}\} = \{z^\star, 0, \dots, 0\}$ is picked and kept fixed. Then, denoting by $\tilde{x}_n^{\text{rec}}$ the generated image at the n^{th} scale using these noise maps :

$$\begin{aligned}\mathcal{L}_{\text{rec}} &= ||G_N(z^\star) - x_N||^2 \text{ if } n = N \\ \mathcal{L}_{\text{rec}} &= ||G_n(0, \tilde{x}_{n+1}^{\text{rec}} \uparrow^r) - x_n||^2 \text{ if } n < N\end{aligned}$$

2.2 Random Sample Generation

SinGAN is a *couteau suisse*, and can be used for a wide variety of tasks, among them :

- Image sample generation
- Super resolution
- Paint to Image
- Harmonization
- Animation
- Editing

In this report we focus on image generation since it is the main task of SinGAN (the others being adaptations of image generation).

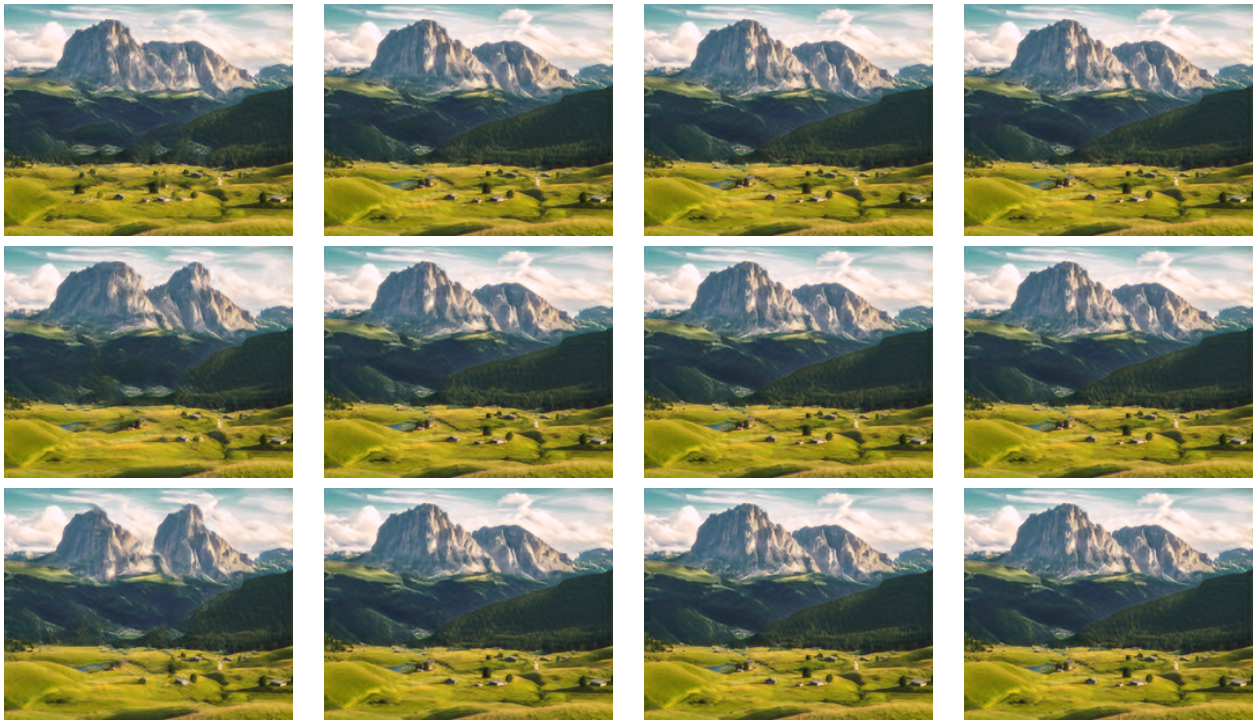
SinGAN allows control over the amount of variability between generated samples, by choosing the scale from which to start the generation at test time.

To start at scale n, we fix the noise maps up to this scale to be $\{z^*, 0, \ldots, 0\}$, and use random draws only for $z_n, ..., z_0$. Starting the generation at the coarsest scale ($n = N$) results in large variability in the global structure. This may lead to unrealistic samples. On the other hand, starting the generation from finer scales enables to keep the global structure intact while altering only finer image features, but this can lead to the generated images being too similar to the original image. Therefore the choice of the generation scale depends on the application and the desired similarity to the original image.

2.3 SinGAN generation examples

We generated sample images with SinGAN to visually assess its performance and to use for our experiments. Below we display examples of generated images at different scales.

Note that the scale notations between the official implementation and the SinGAN paper are inverted, and from here to the end of this report we'll use the implementation's notations because we believe they make more sense, so scale 0 here is scale N in the paper, scale 1 is N-1 in the paper etc.. This makes the generator we're referencing invariable w.r.t. a change in N.



(i) Scale 0 (j) Scale 1 (k) Scale 2 (l) Scale 3

Figure 2: SinGAN generated samples at different scales

We observe that at scale 0, the generated images present some originality, while at scale 1 we only see small changes in minor details and starting from scale 2 the generated images look

almost identical (both to each other and to the original image). This observation remains consistent when changing the training image.

3 The PatchMatch algorithm

3.1 The Original PatchMatch Algorithm

PatchMatch [9] is an algorithm that quickly finds approximate nearest neighbors matches between image patches. Formally, for two images S and T with patches P and Q, it computes a Nearest Neighbor Field (NNF), that is an approximation of the function :

$$\begin{aligned} f &: P \to Q \\ & p \mapsto \text{argmin}_{q \in Q} D(p, q) \end{aligned}$$

Where the function D is a distance between patches, usually the euclidean distance. In practice, the algorithm computes an equivalent offset field as follows:

- *Initialization*: Random initialization of the offsets using a uniform distribution over Q.
- *Iteration : propagation & random search* : The image is raster scanned alternatively from left to right and top to bottom on even iterations, and from right to left and bottom to top on odd iterations to avoid biases. For each pixel, we perform a propagation step and a random search step.

  ◦ Propagation: When the image is scanned from left to right, if we denote (x, y) the position of the pixel scanned, we take the new value for $f(x, y)$ to be the argmin of $\{D(f(x, y)), D(f(x-1, y)), D(f(x, y-1))\}$, where $D(v)$ is the patch distance between the patch at (x, y) in S and the patch at $(x, y) + v$ in T. The intuition behind this is that adjacent pixels are likely to have adjacent nearest neighbors. By doing this, when a pixel finds a "good" neighbour (i.e. with low distance), it is instantly propagated to the whole region next to it: every pixel propagates the good find to the pixel next to it, which propagates it to the next, etc..

  ◦ Random search: Denote (x, y) the current pixel. First we pick a random number $R_i \in [-1, 1] \times [-1, 1]$, then we search for a nearest neighbor among the candidates $u_i = (x, y) + f(x, y) + \text{int}(w\alpha^i R_i)$, with w a large number, usually equal to the maximum image dimension, and $\alpha \in [0, 1]$, and where $(x, y) + f(x, y)$ refers to the current neighbor. We do that for $i = 0, 1, ...$ until the radius $w\alpha^i$ is below 1 pixel. This means that we search for a better neighbor among randomly sampled pixels from squares surrounding the current neighbor, with varying square radii.

The algorithm runs for a given number of iterations, but it can be usually be set under 10, as it usually converges after a few iterations.

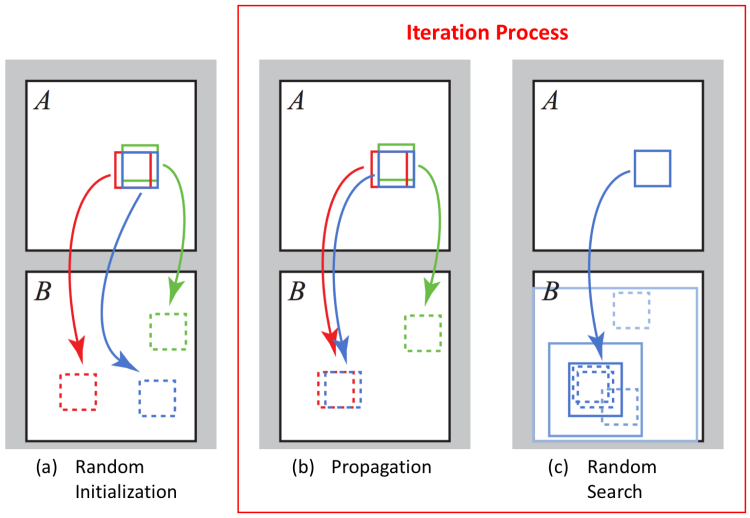


Figure 3: Illustration of the PatchMatch algorithm. Image credit: [9]

3.2 Generalized PatchMatch

A subsequent paper by the same authors [12] proposed an extension of PatchMatch, to deal with rotation and scaling of patches, as well as the use of arbitrary features to compute the distance between patches.

To deal with rotations and scaling, the search space of the original PatchMatch algorithm is simply expanded from (x, y) to (x, y, θ, s), with $\theta \in [0, 2\pi[$ a rotation angle and $s \in \mathbb{R}$ a scale. The NNF hence becomes a mapping $f : \mathbb{R}^2 \rightarrow R^4$. The propagation phase also changes. To deal with the rotations, we propagate using more patches. If we note $T(f(x))$ the transformation defined by (x, y, θ, s), we have using a first order Taylor expansion :

$$T(f(x)) \approx f(x - \Delta_p) + J_T(f(x - \Delta_p))\Delta_p$$

with J_T the jacobian of T.

Therefore, for a given pixel (x, y), we search the minimum distance in :

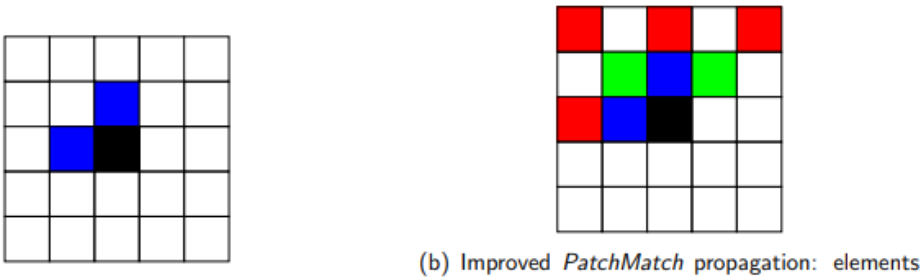
$$\min_{\Delta_p \in \{(1,0),(0,1)\}} \{f(x - \Delta_p) + J_T(f(x - \Delta_p))\Delta_p\}$$

The generalized algorithm preserves the simplicity of the original version, but its computational complexity increases sharply, not only because of the patch interpolation but also because of the higher number of iterations necessary to converge in a 4d search space and the higher chance of being trapped in a local minimum. Empirical tests also show that this method is not very robust to rotation and translation [13]. We will therefore resort to another version of the PatchMatch algorithm described in the next section.

3.3 Modified PatchMatch algorithm

PatchMatch can be used to detect copy-move forgery. A novative model was presented in [8], by Cozzolino et al, which is based on a modified version of PatchMatch to handle rotations more efficiently. A subsequent revision of this model was also proposed by Thibaud Ehret of ENS Cachan [14], and uses the same modified PatchMatch algorithm.

The idea here is simple: the propagation part of PatchMatch is modified, taking into account the 8 pixels shown on figure 4 (right) as neighbor candidates instead of only the 2 pixels considered by the original algorithm (left).



(a) Original *PatchMatch* propagation: elements in blue are propagated to the one in black

(b) Improved *PatchMatch* propagation: elements in blue (original propagations), green (completed zero order propagations) and red (first order propagations) are propagated to the one in black

Figure 4: Modified propagation step. Image credit:[14]

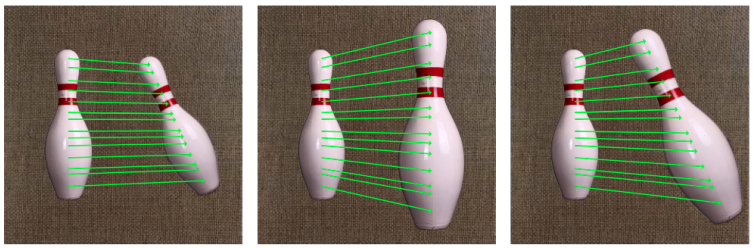


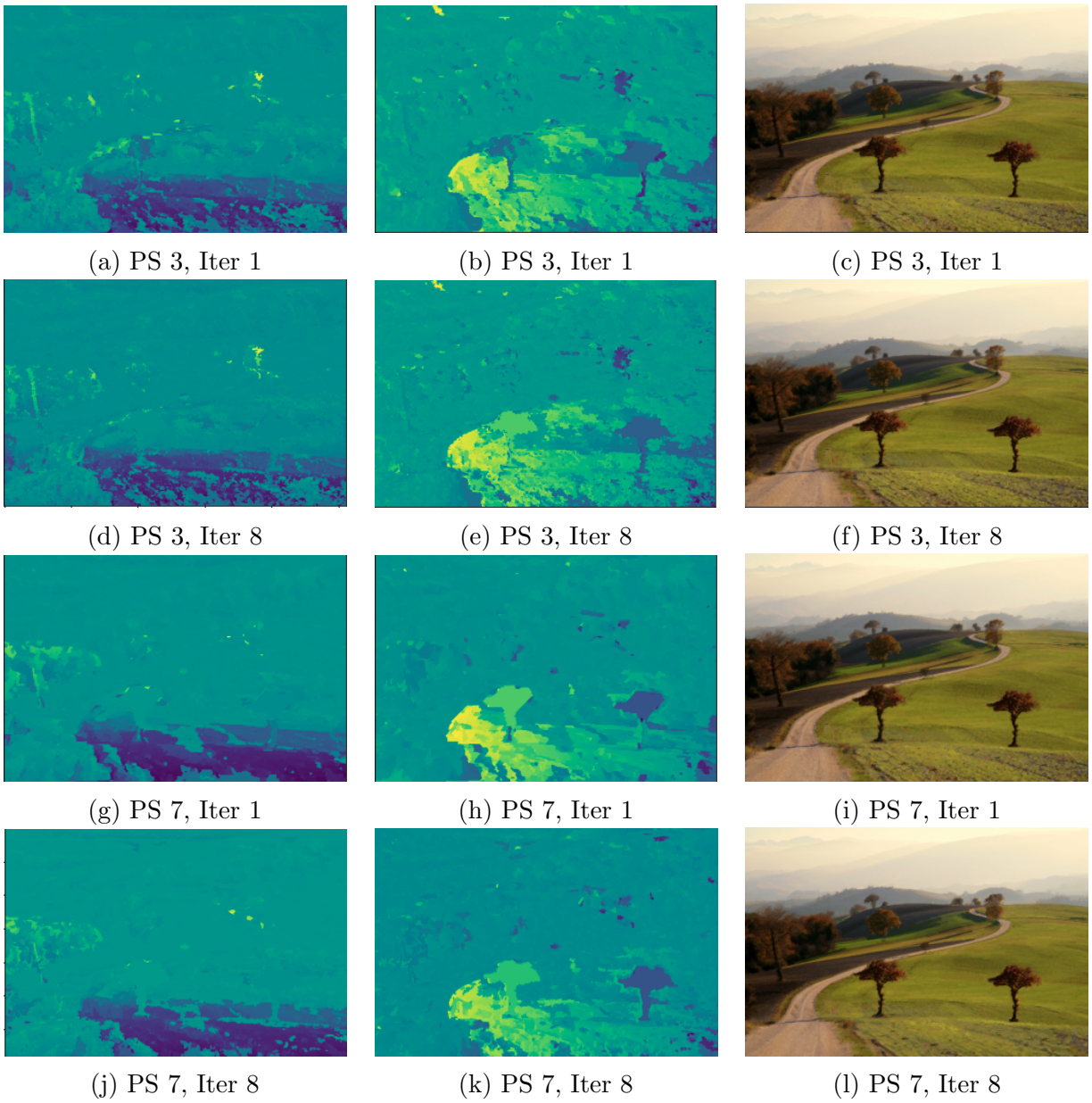
Figure 5: Linearly varying offsets are robust to scaling and rotation. Image credit:[8]

Combined with the scale/rotation invariant features presented in the next section, this simple modification of the PatchMatch algorithm allows it to detect scaled and rotated patches with only a slight increase in the computation time, since the random search phase in unchanged.

3.4 PatchMatch experimentation and visualization

Here we test the PatchMatch algorithm on an example image containing copy-move forgery:

Figure 6: Original image



(a) PS 3, Iter 1 (b) PS 3, Iter 1 (c) PS 3, Iter 1

(d) PS 3, Iter 8 (e) PS 3, Iter 8 (f) PS 3, Iter 8

(g) PS 7, Iter 1 (h) PS 7, Iter 1 (i) PS 7, Iter 1

(j) PS 7, Iter 8 (k) PS 7, Iter 8 (l) PS 7, Iter 8

Figure 7: PatchMatch outputs: row offset (left), column offset (center), recreated image (right) for different patch sizes (PS) at different iterations (Iter)

We invite the reader to consult our notebook for the full results at every iteration. What we can observe is that the recreated images are visually very similar to the original image since the first iteration, although the pixel-level details were blurred and became more detailed gradually. For patch size of 3, The algorithm needed 7 iterations to find and propagate the correct neighbors of the trees. For patch size of 7, the two trees converge to each other since the first iteration, but the final recreated image is slightly more blurry than with patch size=3. We also observe larger uniform regions in the offset map, and those regions might be detected as false positives by the copy-move detection algorithm. We conclude that the choice of the patch size involves a trade-off between larger patches which converge faster but may lead to more false positive detections, and smaller patches that lead to a more fragmented offset field, therefore fewer false positives but more false negatives (undetected).

4 Copy-Move Detection

In this section we present our implementation of the copy-move forgery detection algorithm proposed in [8] (we will refer to this paper as the reference paper). The algorithm consists in 3 steps: preprocessing the image into features, using PatchMatch to obtain the offset field, then postprocessing the offset field to detect its uniform zones as copies.

4.1 Features used for the computation of distances

The features used for the computation of the distance between patches can be the RGB colors (3 features), but their performance may be severely affected by JPEG compression, noise addition, and other common distortions. In the literature, to improve robustness, features are typically extracted through some transforms, like DCT, Wavelet, PCA, or SVD, but such features are sensitive to rotation and scaling. The reference paper suggests Zernike Moments (ZM), Polar Cosine Transform (PCT) and Fourier-Mellin Transform (FMT) as potentially well-performing features and compares them, and we can conclude from their experiments that ZM and PCT both perform equally well. For this project we chose to use PCT features, but we note that ZM features are a viable alternative.

For every couple of parameters (n,m), the PCT feature $F_{I(s)}(n,m)$ of patch I taken from a single-channel image is defined by the scalar:

$$F_I(n,m) = \int_{\rho=0}^{\infty}\int_{\theta=0}^{2\pi} \rho \bar{R}_n(\rho) \times \frac{1}{\sqrt{2\pi}} I(\rho,\theta) e^{-jm\theta} d\theta d\rho$$

with $R_n(\rho) = C_n \cos(\pi n \rho^2)$ and $\bar{R}_n(\rho)$ its complex conjugate.

We will use 10 couples of (n,m) parameters, resulting in 10 features for every pixel in the image. We note that the features are comprised of 3 terms: R_n (the cos term), the exponential term, and the image patch term. The integral will be approximated by a sum.

The idea to calculate those features is the following: Let (n,m) be the PCT parameters. First we convert the image to grayscale to obtain a 1-channel image. For every pixel s in the image, define its 8x8 surrounding patch $I(s)$. We create a 8x8 matrix defining $R_n(\rho)$, another one defining $e^{-jm\theta}$, then we multiply them element-wise. We obtain a filter (8x8 matrix) that we will convolve on the image, i.e. we will multiply by the patch $I(s)$ for every pixel s in the image.

We provide in the notebook an illustrated description of how the features are computed, and in Fig. 8 the 10 filters convolved on the image, as well as the resulting features in Fig. 9.

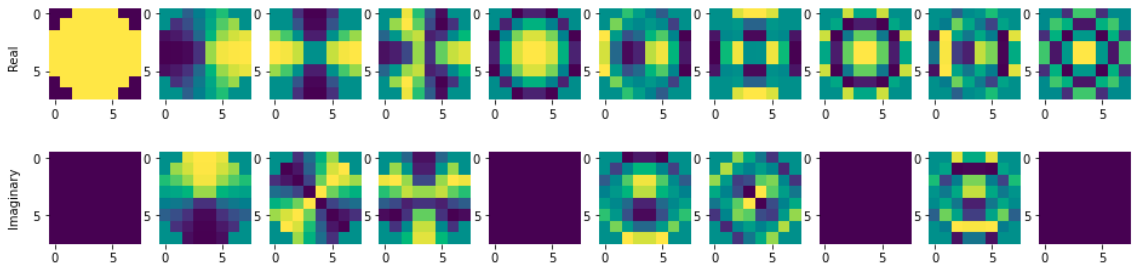


Figure 8: PCT filters

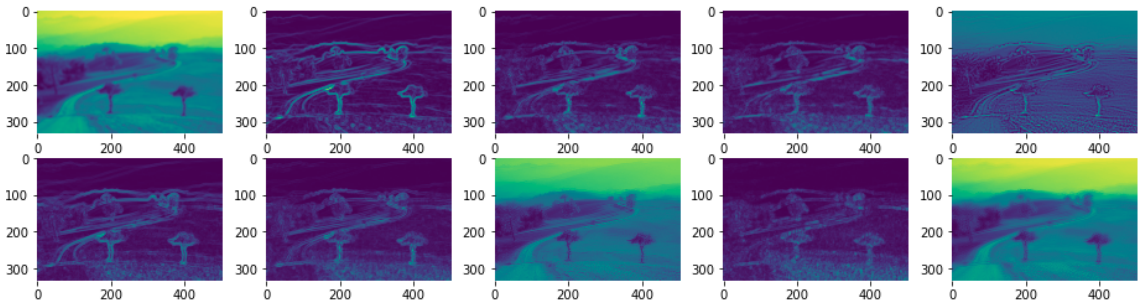


Figure 9: PCT features

The features capture the different properties of the image: edges, uniform zones, etc.. And they can be passed to the PatchMatch algorithm (as a 10-channel image) which is what the article proposes. However, we experimented with different feature combinations to check which ones work better, like using the previously illustrated features computed from the grayscale image ($\rightarrow$ 10 features), using the features computed from each of the 3 channels of the original image ($\rightarrow$ 30 features), using the original 3-channel image instead of calculating PCT features ($\rightarrow$ 3 features), and using a combination of the 10 PCT features (from the grayscale image) and the 3 RGB channels of the original image ($\rightarrow$ 13 features). After optimizing the parameters of the detector for each of those possible features, we concluded that using 30 features works best (yields the least number of false positive detections and false positive undetected copies), although all of them worked decently. Since the purpose of this project is to analyze the patch copies of SinGAN and not to propose a fast detection algorithm, we opt for using 30 features to get the best performance. It is noteworthy however that when using the original RGB image, we need to use a PatchMatch patch size of about 5 to get good performance, so every patch distance calculation involves 25 pixel distances. However when using the PCT features, those features already contain the needed information about the 8x8 patch surrounding a given pixel so we can use a patch size equal to 1 (which works the best actually), so every patch distance calculation involves only one pixel. As a result, despite using 30 features the CPU time of PatchMatch remains comparable to using just the original 3 RGB channels, and we get better performance.

4.2 Postprocessing the offset field

After computing the features, we obtain a 30-channel image on which we run the modified PatchMatch algorithm and obtain the offset field. Afterwards, some post-processing steps are applied in order to detect uniform regions in the offset field as precisely as possible, while avoiding false detections. We present below the retained post-processing methodology:

1. Regularize the offset field with a median filter
2. Convert the offset field to neighbor field (add pixel coordinates)
3. Calculate the DLF error: Difference of 4 convolutions for each neighbor field
4. Mask the pixels having DLF error below a given threshold
5. Eliminate pixels where the offset distance is below a given threshold

6. Remove small objects
7. Mirror detected objects
8. Remove small objects again
9. Add borders to the mask
10. Binary dilation

To illustrate the postprocessing methodology, we provide a step-by-step walkthrough of the applied operations. We start with the offset field obtained from PatchMatch:

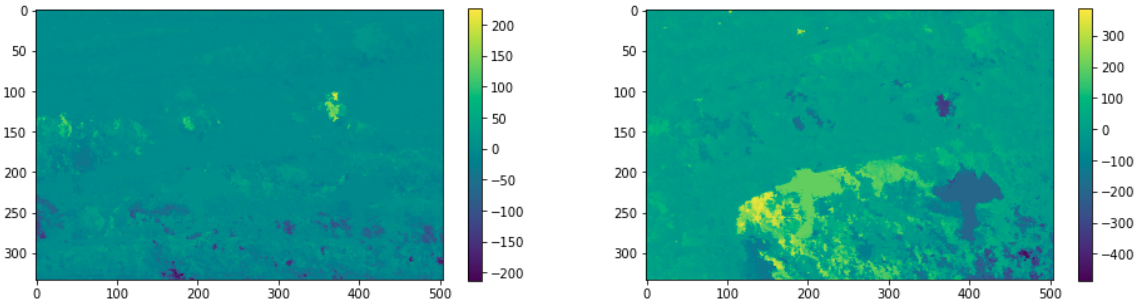


Figure 10: Row (left) and column (right) offset fields

1. Regularize the offset field with a 4×4 median filter. This regularization might seem counter-intuitive as our goal is to distingush uniform and non-uniform regions in the offset field, and regularization makes all the field more uniform, but it actually helps to make semi-uniform regions of rotated or scaled objects uniform. Without regularization, most rotated or scaled duplicates aren't detected.

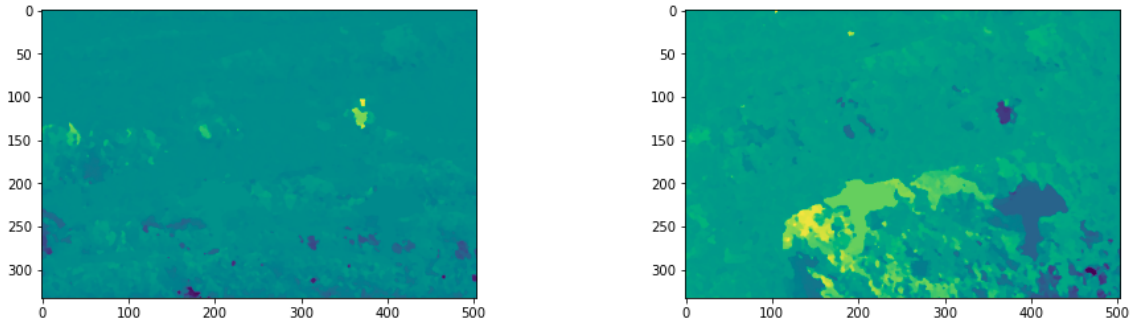


Figure 11: Offset fields after regularization

2. Add the coordinates of the pixels to the offset to obtain the coordinates of the neighbors

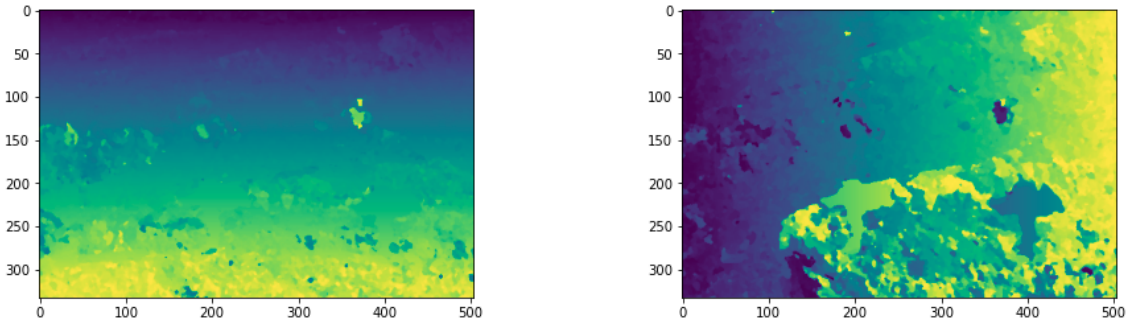


Figure 12: Neighbor fields

3. Compute DLF (Dense Linear Fitting) error

The reference paper defines the DLF error for a pixel s in an N-pixel neighbourhoud (for example the 8 or 18 pixels surrounding it) by: $\widehat{\delta}(s_i) = As_i, \quad i = 1, \ldots, N$ where A is an affine transformation set so as to minimize the sum of squared errors w.r.t. the true offsets:

$$\epsilon^2(s) = \sum_{i=1}^{N} \left\| \delta(s_i) - \widehat{\delta}(s_i) \right\|^2$$

The idea here is to to estimate for every pixel what is its offset using the offsets of its surrounding pixels as covariates in an affine regression, and the DLF error is then defined by the error of this estimation. Therefore the DLF error is low in uniform regions and high in non-uniform regions, and this allows us to detect uniform regions in the offset field.

The reference paper proves that the DLF error can be approximated using the formula $\epsilon^2(s) = \left(\delta^T \delta\right) - \left(\delta^T q_1\right)^2 - \left(\delta^T q_2\right)^2 - \left(\delta^T q_3\right)^2$ where $\delta = [\delta(s_1), \delta(s_2), \ldots, \delta(s_N)]^T$; and q_1, q_2, q_3 are vectors independent of the offset field. This allows us to greatly reduce the computation cost of the DLF error, by computing it as a difference of convolutions using a filter defining the N pixels included in the neighborhood δ and 3 filters defining q_1, q_2, and q_3:

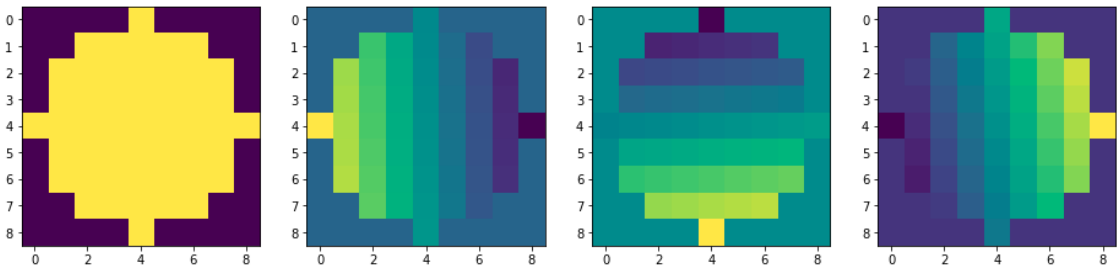


Figure 13: Filters used to calculate the DLF error

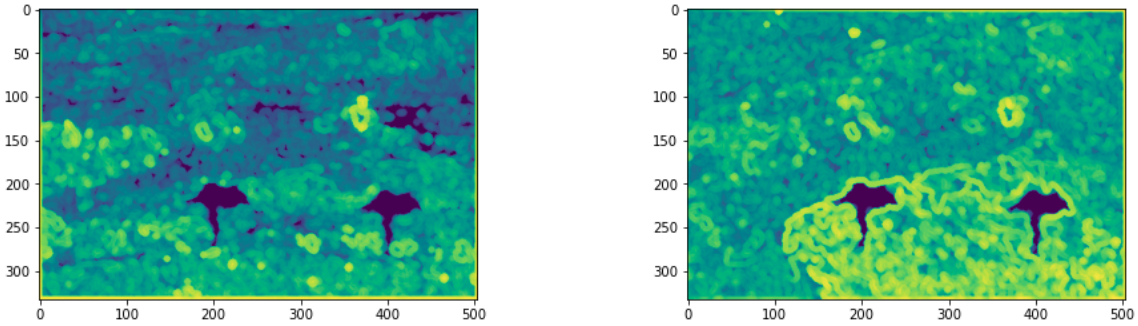


Figure 14: DLF error of the row and column offset fields

The two DLF errors (of the row and column offset fields) are then summed.

4. Mask the pixels having DLF error below a given threshold (50):

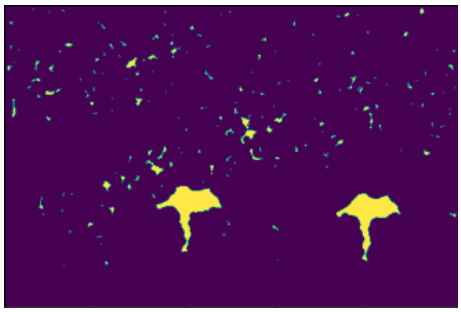


Figure 15: Binary mask obtained by applying a threshold on the DLF error

5. Eliminate pixels where the offset distance is below a given threshold (40 pixels):

Figure 16: Detection mask after applying a threshold on the offset distance

6. Remove small objects (having a total connected area of less than 2.5% of the area of the image):

Figure 17: Detection mask after removing small objects

7. Mirror detected objects : if a pixel is detected, its neighbor is detected

Figure 18: Detection mask after mirroring the detections

8. Remove small objects again (total connected area of less than 2.5% of the area of the image):

Figure 19: Detection mask after removing small objects

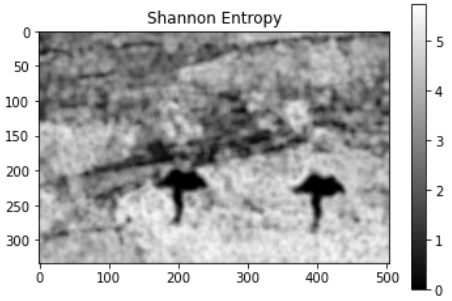
9. Add borders to the mask: This is because when we compute the PCT features using convolutions, the borders can present artifacts and should be removed. Here we pad the mask with zeros so that it has the same size as the original image.

10. Apply binary dilation to the mask: DLF error crops some of the duplicated area's borders, and binary dilation allows us to revert this effect

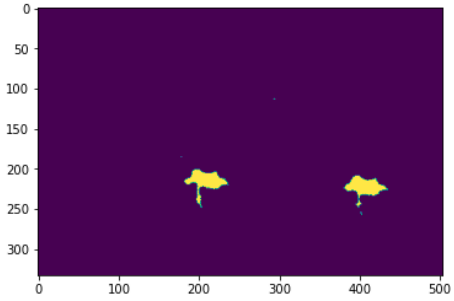
Figure 20: Detection mask after binary dilation

After these steps we obtain our final detection mask highlighting the duplicated objects in the image.

We note that we also explored some other interesting methods to detect uniform regions. Shannon entropy filter was one of the most promising methods:



(a) Shannon entropy applied on the sum of absolute offset fields



(b) Detection mask obtained by thresholding the Shannon entropy

The downside of this method is that it is more sensitive to the threshold than our main method and doesn't work as consistently.

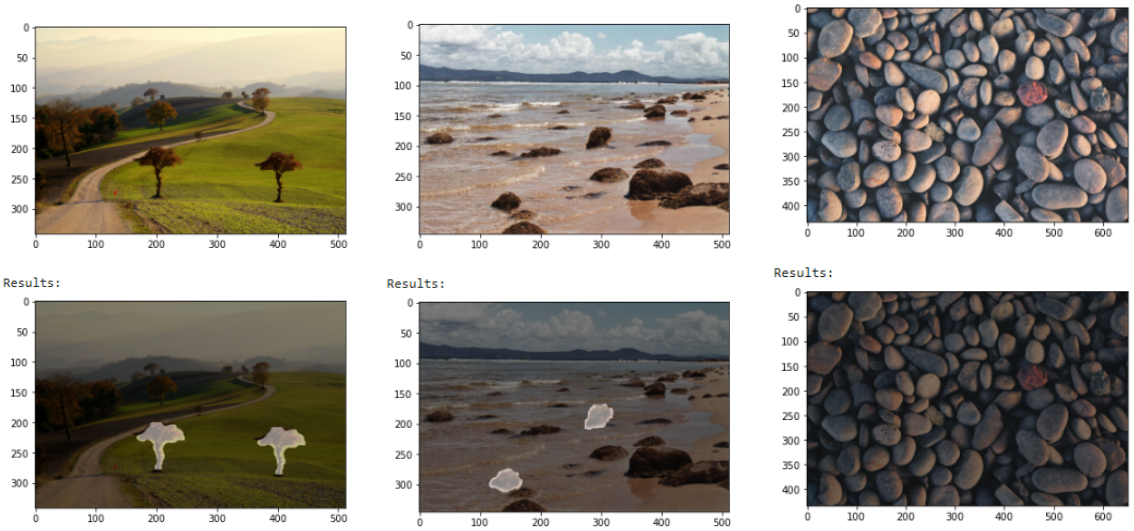
We also note that to choose the algorithm's parameters, we decided to optimize them manually because the parameter space is large and performing a grid-search would take a long time. Our main remarks from the optimization process is that the algorithm is quite sensitive to the choice of the parameters and if some parameter is altered, many others need to be changed too to get the algorithm to work again. The parameters that have the biggest impact on the algorithm's performance are the DLF radius, DLF threshold and median filter size. And the same parameters might not work perfectly for all image sizes. For example when dealing with extremely small images, it might be necessary to either adjust the parameters or just upscale the images in order to get good performance.

5 Experiments and results

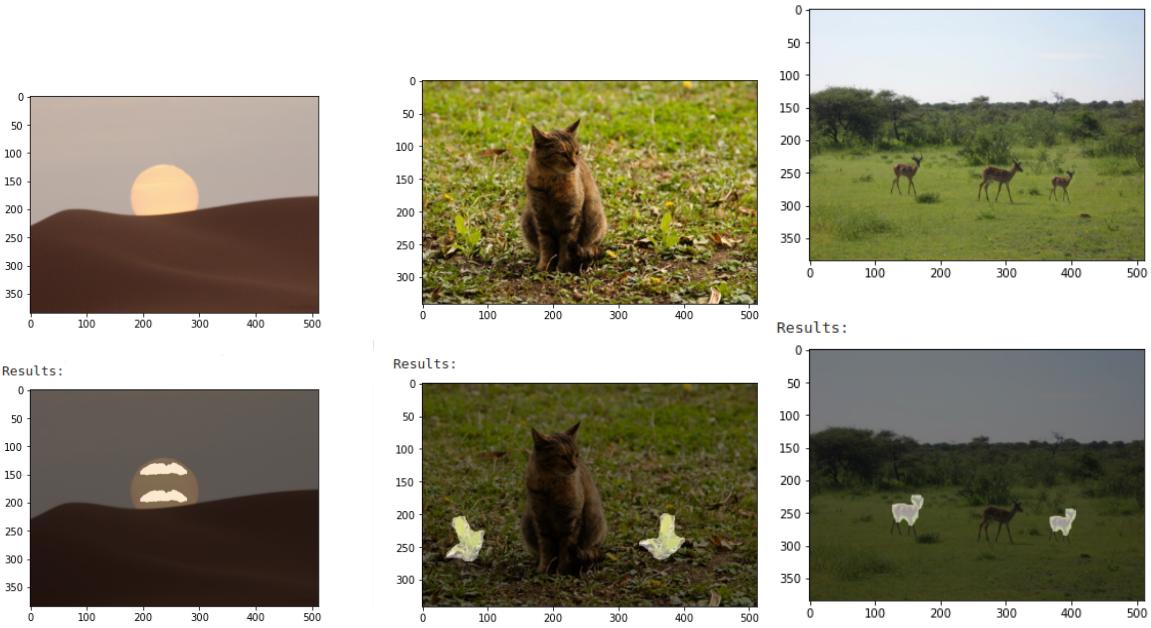
5.1 Copy-move forgery detection

We implemented the algorithm described in the previous section, which given an input image, outputs a binary mask highlighting the duplicated regions in the image. To practically evaluate its performance, we tested it on various challenging images.

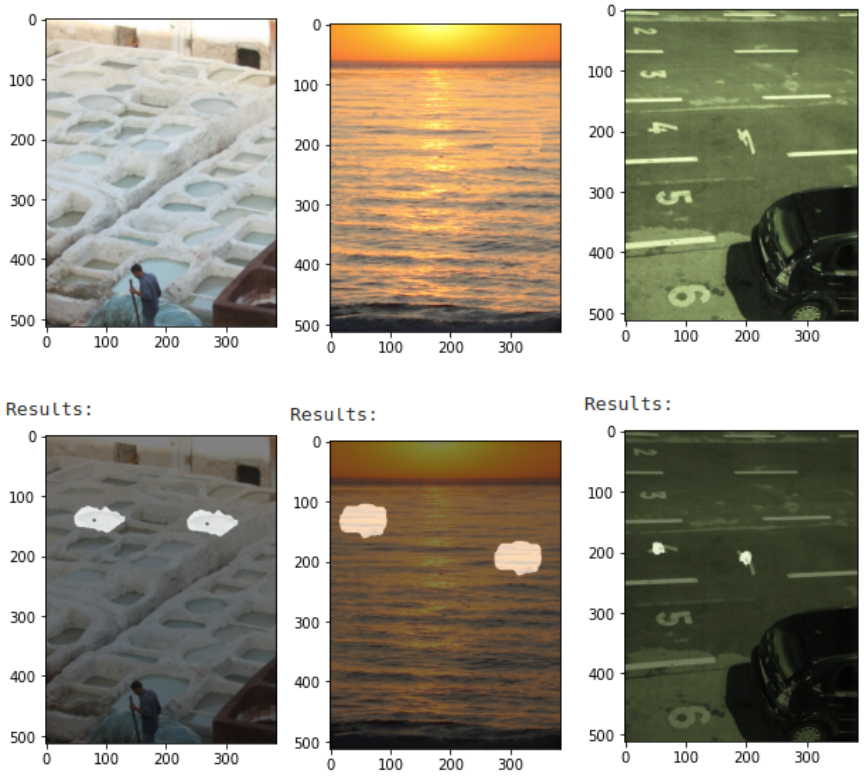
5.1.1 Detecting duplicates in a single image



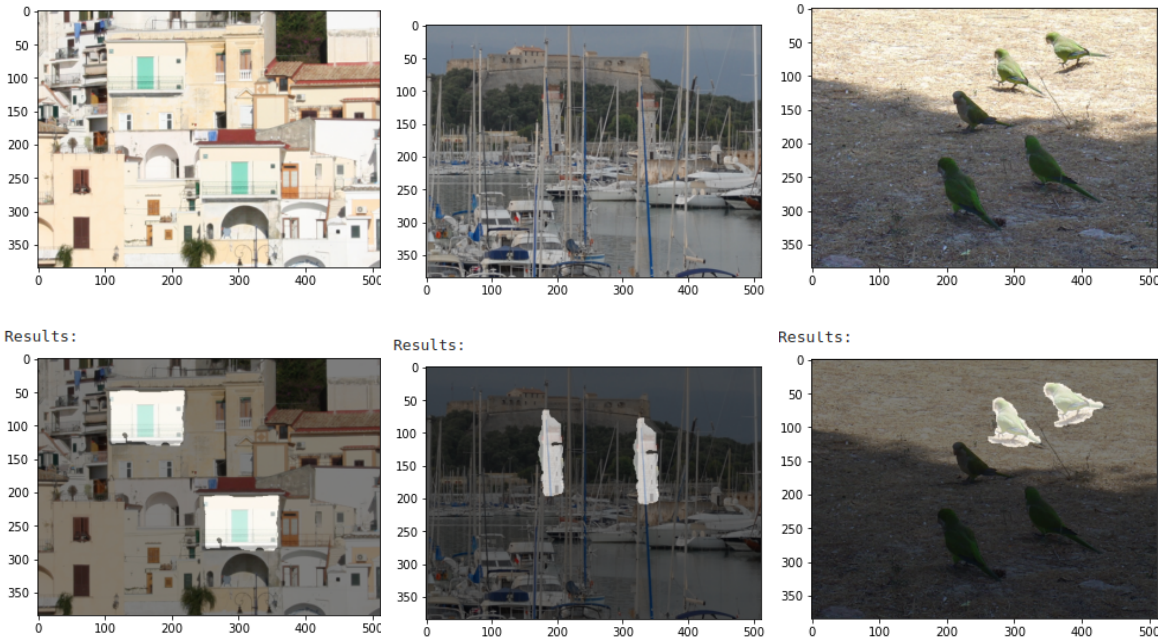
(a) Translated duplicate (b) Rotated duplicate (c) No duplicates in this image but many similar objects



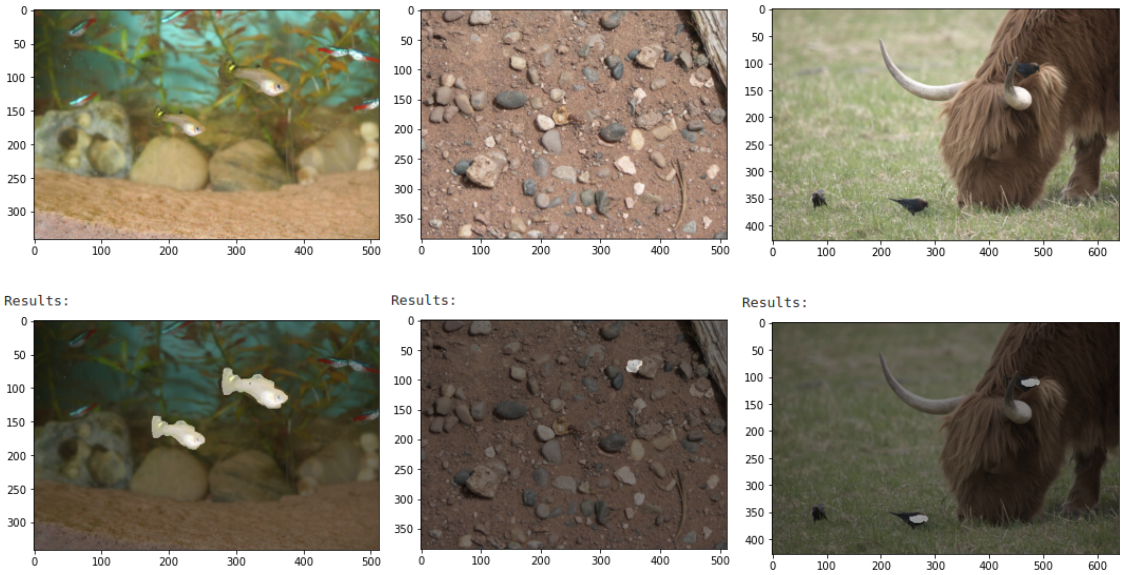
(a) Translated duplicate (b) Rotated duplicate (c) Scaled duplicate



(a) Translated duplicate (b) Translated duplicate (c) Rotated duplicate



(a) Translated duplicate (b) Translated duplicate (c) Rotated duplicate



(a) Scaled duplicate (b) Scaled duplicate (c) Translated duplicate

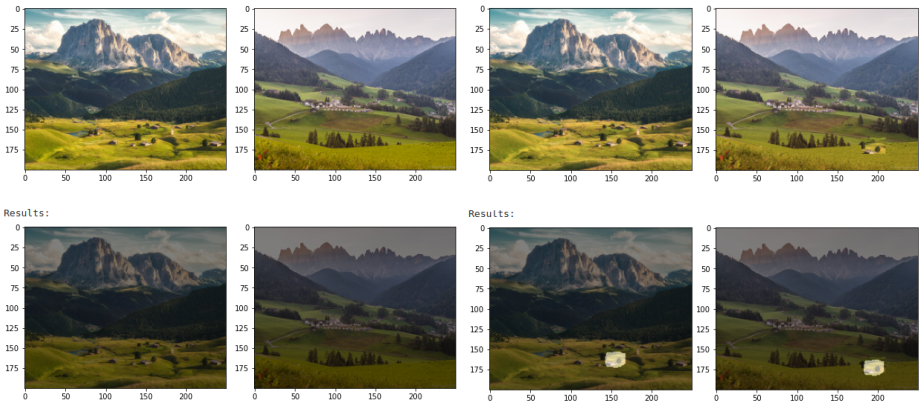
Figure 26: Detection of duplicates in an image. The copies detected are highlighted in white

We see that the algorithm successfully detected all the duplicated objects even when they are rotated or scaled (except for one image, image b in the last row, where only one of the duplicates was detected), and that it doesn't return false positives (false detections). These results are very satisfying.

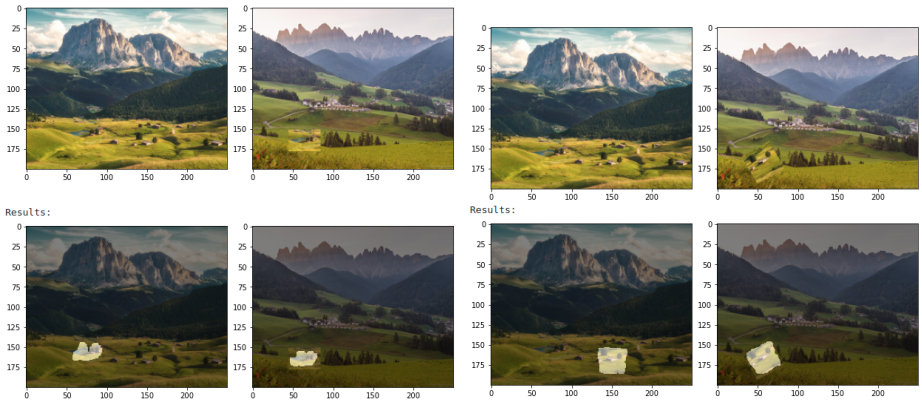
5.1.2 Detecting duplicates between two images

Our previous implementation allows us to detect duplicates in one image, but for our SinGAN copy detection experiments we will need to detect duplicates between two images. To do this, a first possibility was to use the two images as input for PatchMatch while keeping the same preprocessing and postprocessing steps. However this method didn't work (returns too many false positives, see notebook for illustration). This is because different images usually have different morphological stuctures, and sometimes a macroscopic structure in one image only finds a close neighbour in one specific small patch, so the whole region converges to that single patch which results in large uniform areas in the offset field.

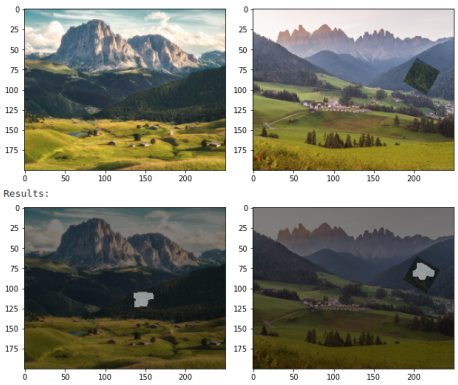
To solve this problem, we converted the two images to one image by simply stacking (concatenating) them and using the detection algorithm on the stacked image. This solution worked well when we tested it on example images (Fig. 29). Although the detection masks are not perfect, copied patches were correctly detected with no false positives.



(a) No duplicate (b) Translated duplicate



(a) Scaled duplicate (b) Rotated duplicate



(a) Rotated duplicate

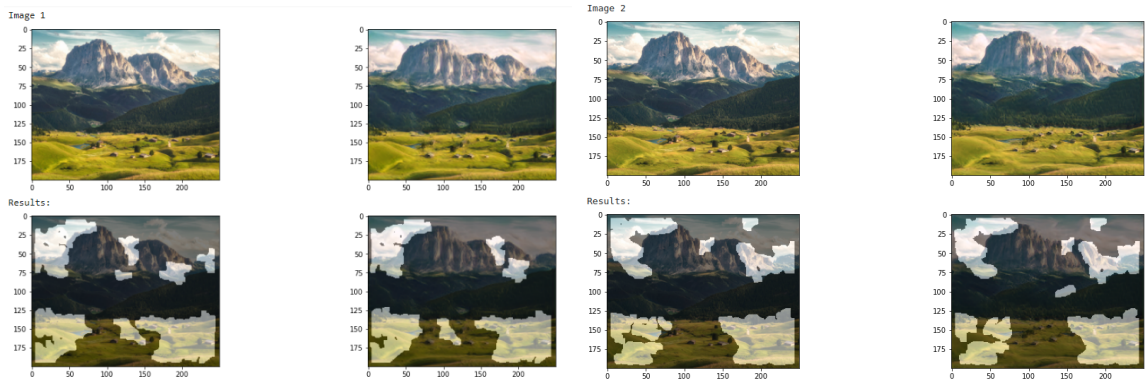
Figure 29: Detection of duplicates between two images

5.2 SinGAN copied patch detection experiments

The experiments below are performed on sets of 50 images for each scale, generated from a single training image using SinGAN with N=7 generators.

5.2.1 Images generated at scale 0

Here we consider images generated at scale 0, which have the most variability and are the most visually different from the original image.



(a) Sample 1 (b) Sample 2

Figure 30: SinGAN copied patch detection at scale 0 for two samples. The original image is on the left of the samples

The detection results for all 50 images are available in the notebook. We observe that a relatively large area of the SinGAN-generated images is detected as a copy. This is because our patch copy detector is robust to transformations such as scaling and rotation. If we didn't detect such transformations, we wouldn't detect anything because SinGAN doesn't perform simple translations that keep keep pixel values identical, but it deforms the image as we can see in the examples above.

Visually, the detected regions in the SinGAN images and the original image are very similar, which indicates that our detection algorithm worked as expected.

We compute some statistics on the patches of the 50 images generated at scale 0:

The average detected region is 32.2% of the area of the image, i.e. 1/3 of the generated image area is copied (with slight deformations) and 2/3 is synthetic. This result seems coherent with the level of similarity that we can visually observe in the generated images.

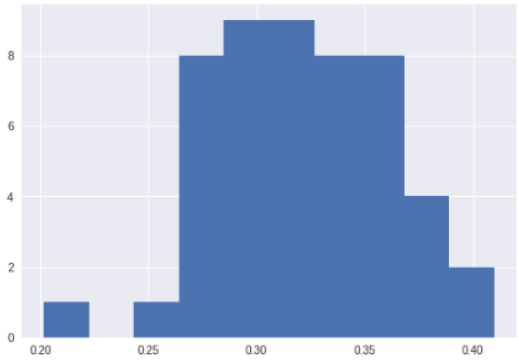


Figure 31: Distribution of the proportion of copied patches

The copy rates of the 50 SinGAN images vary between 20% and 41%, and a 95% confidence interval of the mean copy rate is $[31.0\%, 33.4\%]$.

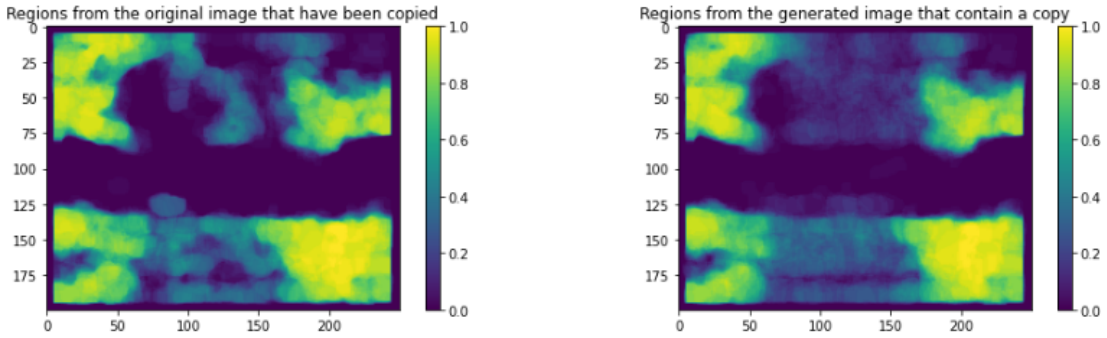


Figure 32: Spatial distribution of the proportion of copied patches

We observe in figure 32 that the distribution of copied regions is not uniform: Some areas tend to be copied much more than others. We can conclude that SinGAN takes more liberty modifying and deforming texture-like homogeneous areas such as the dark forest in the middle of the image (and to a lesser degree the mountains), while areas that contain unique details are more likely to be copy-pasted.

5.2.2 Images generated at scale 1

Images generated at scale 1 have less variability and look visually similar to the original image.

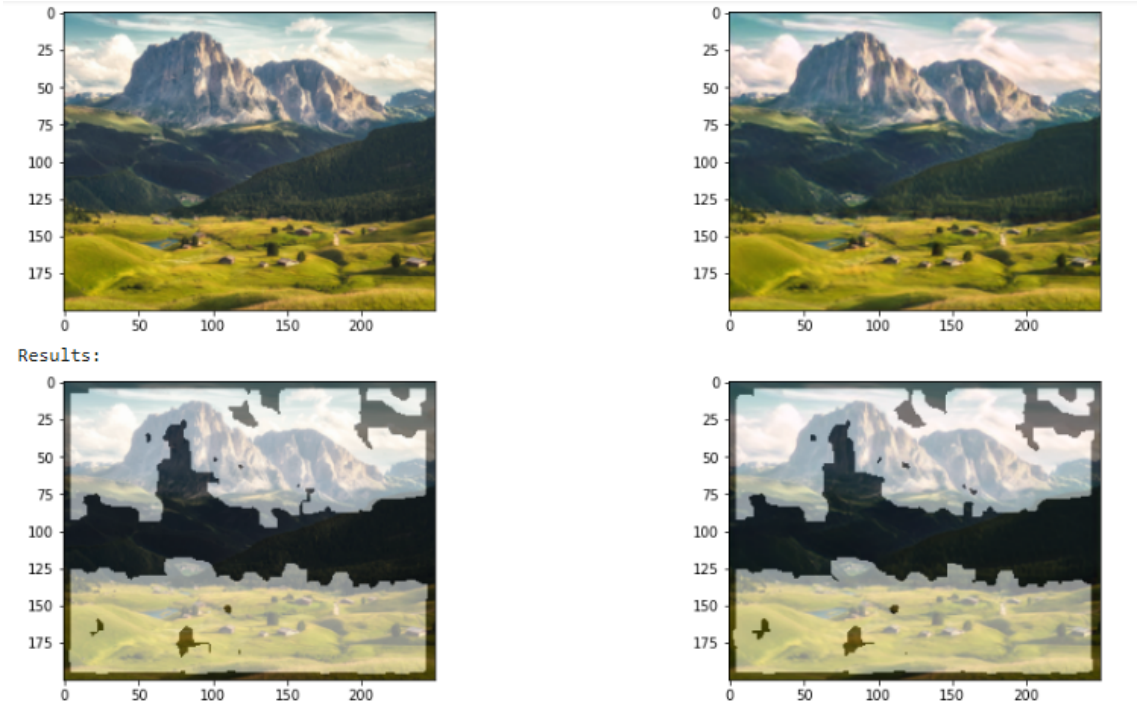


Figure 33: SinGAN copy patch detection at scale 1

We compute the same statistics as before:

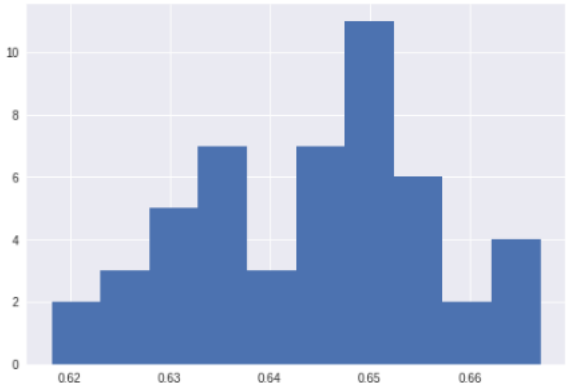


Figure 34: Distribution of the proportion of copied patches for images generated at scale 1

The average detected region is 64.4% of the area of the image, i.e. 2/3 of the generated image area is copied. That is twice as much as with images generated at scale 0.

The confidence interval is much tighter now : $[64.0\%, 64.7\%]$, and we obtain an almost binary spatial distribution of the detections: some regions are almost always copied while others are almost never copied.

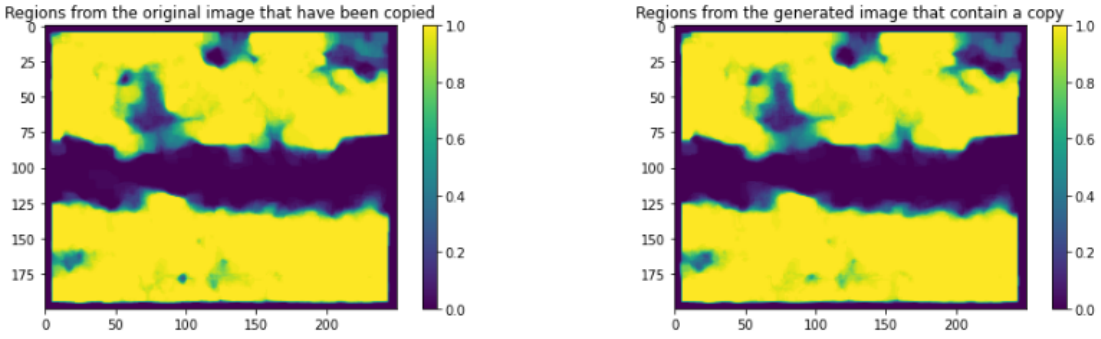


Figure 35: Spatial distribution of copy rate in images generated at scale 1

We obtain very interesting results: as expected, most of the image area is detected, but interestingly the forest is never detected as a copy. At first we thought that the detection algorithm had a problem detecting dark areas, but this is not the case since there are also some regions in the clouds and in the mountain that are also never detected.

The common point in the the regions that were never (or rarely) detected as copies is that they contain a semi-uniform texture with subtle pixel-level details that can only be seen when we zoom in, and what SinGAN tends to do in this case is to dilute those areas and erase the details. We can confirm this by zooming on the images:

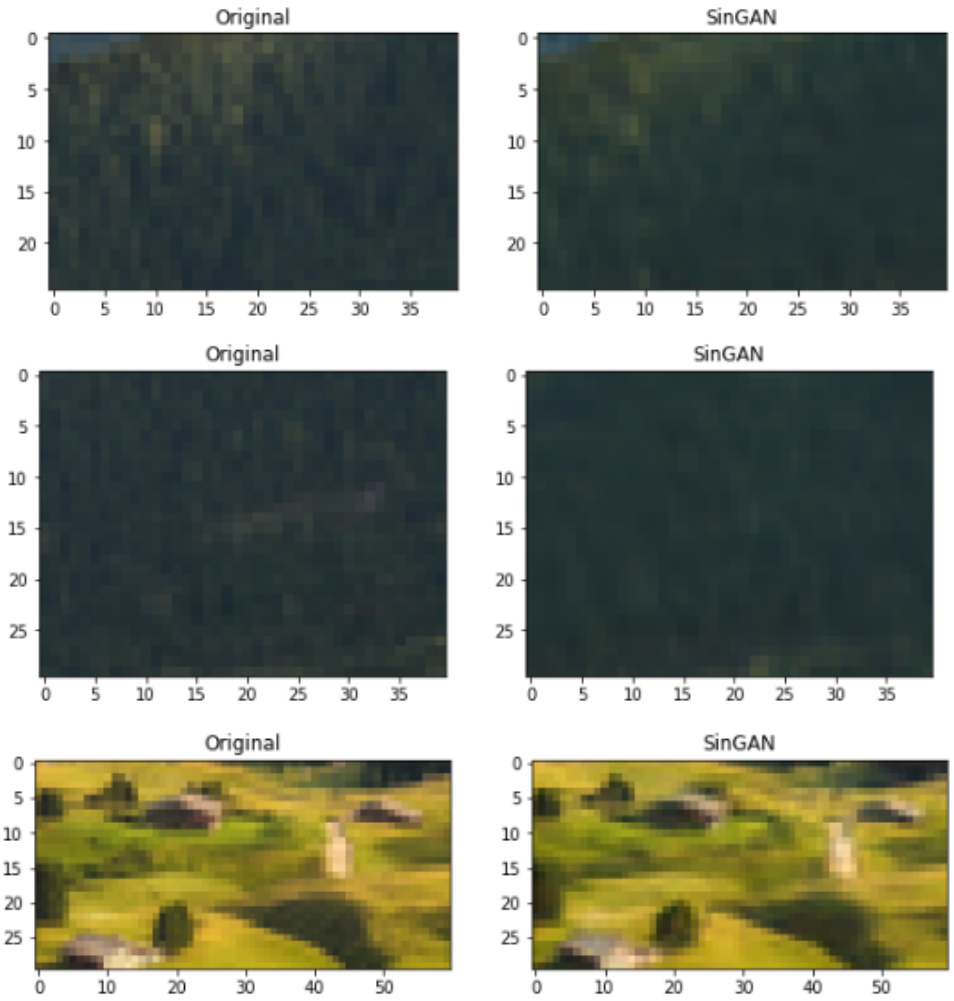


Figure 36: Comparison between original and generated patches

For the first two subfigures the difference between the images is noticeable. SinGAN erased pixel-level details, and generates an almost uniform area. This is why such areas are not detected as copies.

In the bottom figure depicting a non-uniform region it is hard to notice a difference between the two patches, and as such they are detected as copies.

This is an interesting finding: SinGAN tends to dilute the original image and blur its pixel-level details, and this effect is much stronger and more noticeable in homogeneous regions.

5.2.3 Multiscale comparison

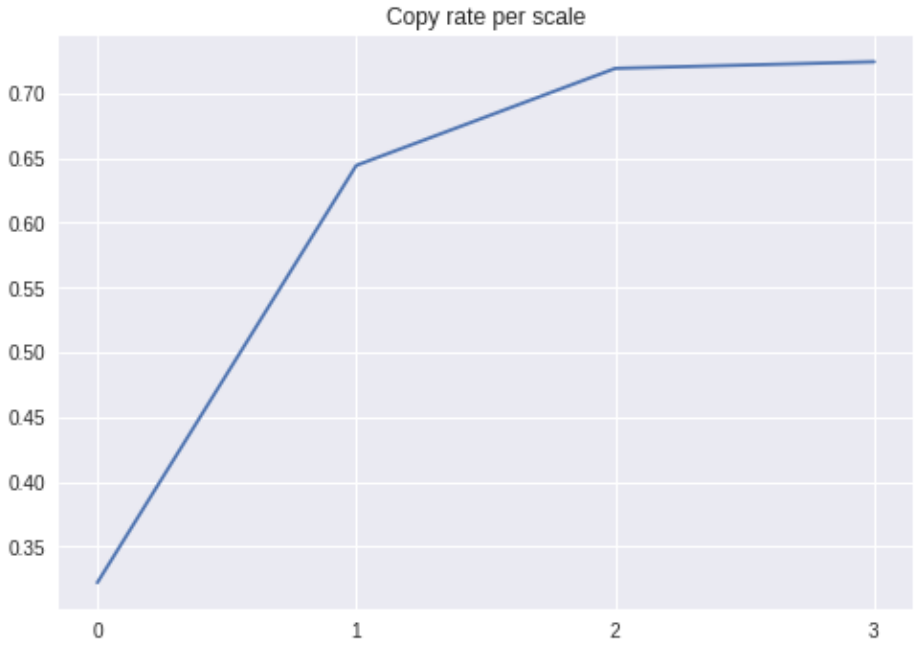


Figure 37: Percentage of copied patches for scales 0 to 3

We can conclude that for this image, the SinGAN-generated images at scale 1 or higher are mostly copied from the original image. Only scale 0 achieves a reasonable level of originality (68%).

5.2.4 Comparison with another image

We rerun the experiments with a different image to assess the variability of SinGAN's originality with respect to the training image.

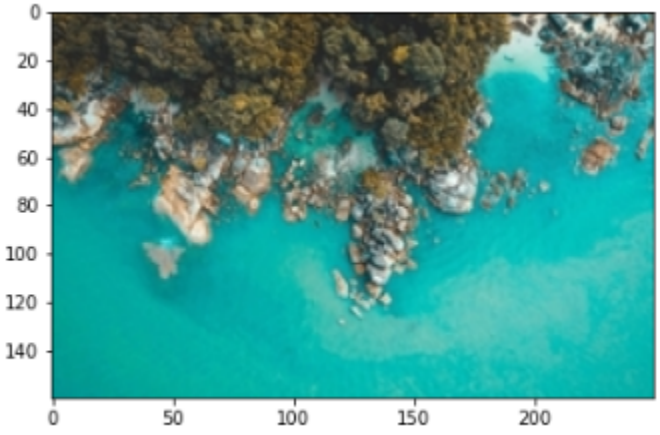
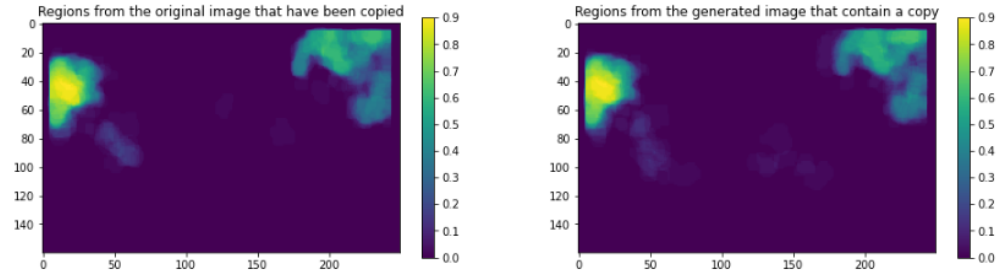
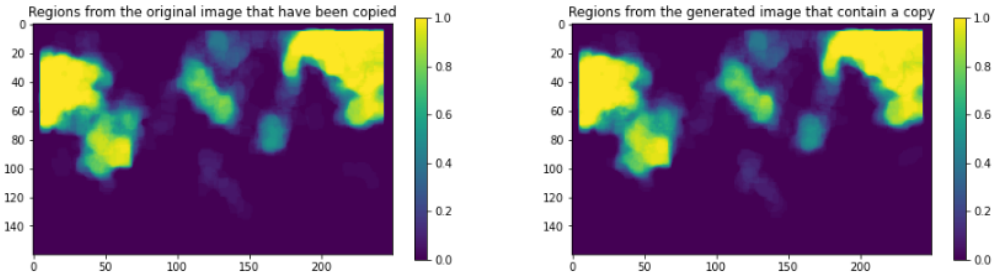


Figure 38: Second training image

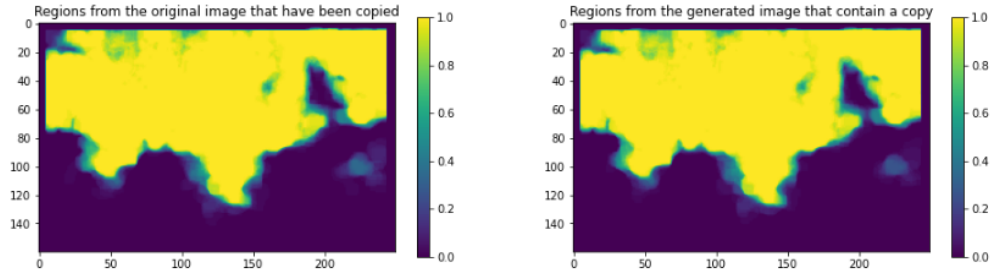
For 5 images generated for every scale, we obtain the following distribution of copied patches :



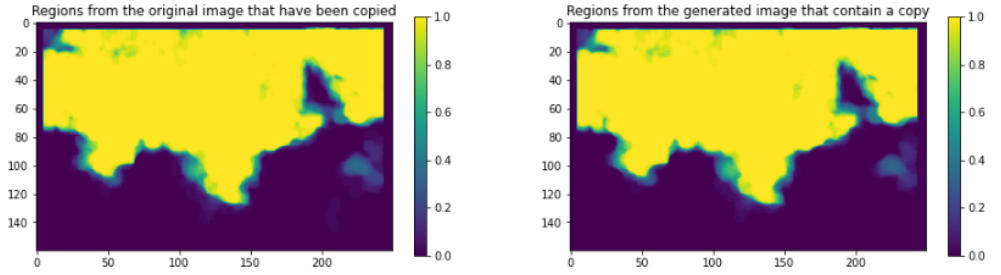
(a) Scale 0



(b) Scale 1



(c) Scale 2



(d) Scale 3

Figure 39: Spatial distribution of copied patches for images generated at different scales

We can confirm that images generated at scale 0 display a rather satisfying level of originality:

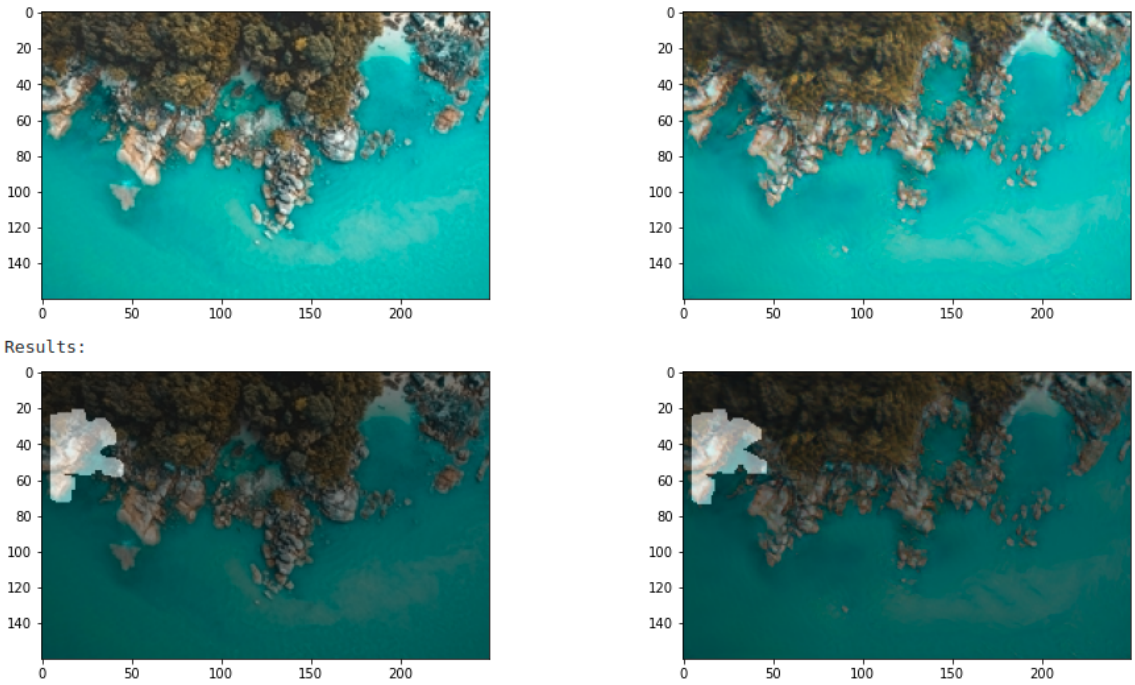


Figure 40: Original and generated image at scale 0

But for scale 3 almost all the non-uniform area is copied :

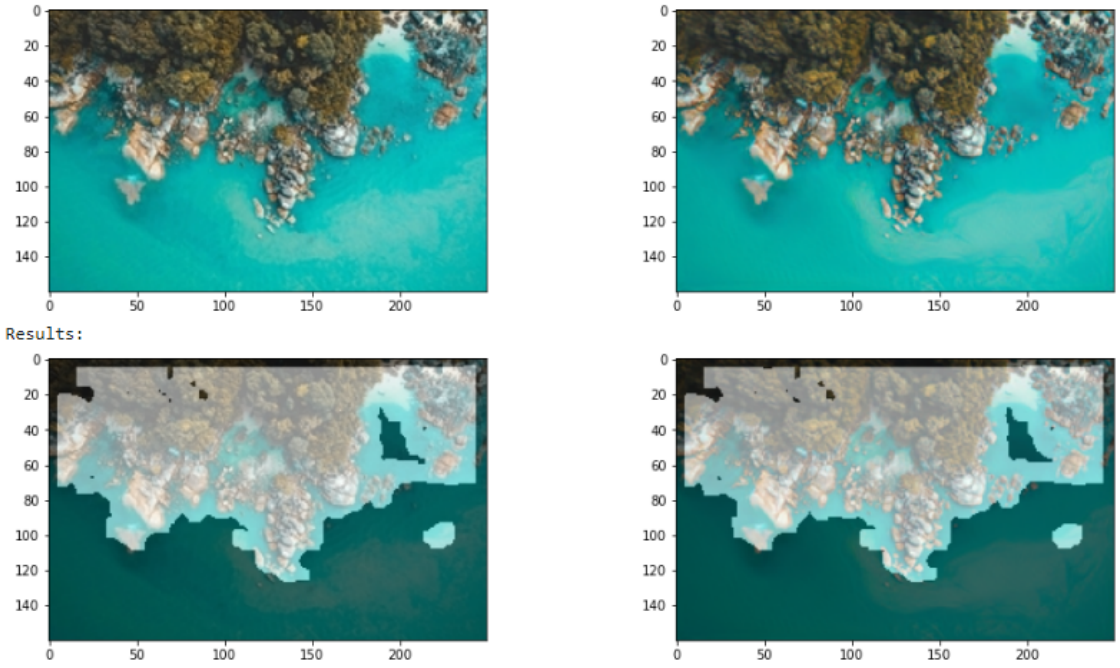
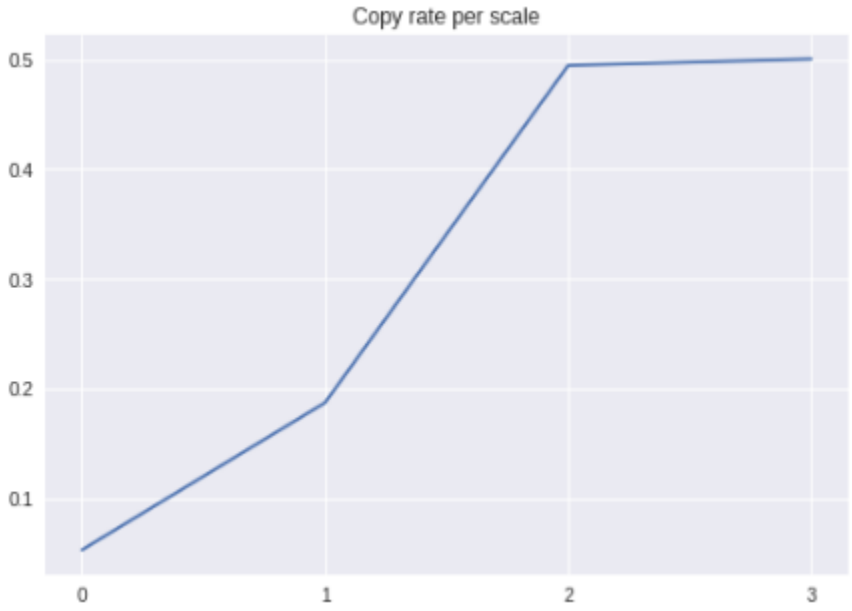


Figure 41: Original and generated image at scale 3

Like in the first image, semi-uniform areas (the water) are diluted and become much more uniform (here we can notice this effect without zooming). Those areas are therefore not detected as copies. As for the land area, it is mostly synthetic for scale 0 and 1 but as we increase the scale, more and more of it becomes copied.

Finally, the percentage of copied patches for every scale is the following :



We conclude that the copy rate of SinGAN is not constant but depends on the image (for this image, the copy rate at scale 0 is 83% lower than the first image). We also see that for both images, the copy rate reaches a plateau after scale 2 and the detection distribution becomes nearly binary: the non-uniform regions are always copied and the semi-uniform regions are never copied (at least not exactly: they get diluted).

6 Conclusion

In this project we explored SinGAN, a novel image-generation algorithm based on generative adversarial networks, and we implemented a copy-move detection algorithm based on PatchMatch and used it to provide a quantitative measure of the originality of the images generated by SinGAN. This project allowed us to assess both the performance of the detection algorithm and the generative capabilities of SinGAN. Our first takeaway is that PatchMatch is a surprisingly effective algorithm that seemed to always find the nearest neighbors in fewer than 10 iterations. We also conclude that the copy-move detection algorithm proposed by Cozzolino et al. delivers very satisfying performance and proves to be robust to transformations such as rotation and scaling. Besides, our experiments on SinGAN-generated images highlights a variable copy rate depending on the training image that can be as low as 5% for some images and as high as 32% on others at scale 0. We also note that starting from scale 1 and above, the copy rate greatly increases compared to scale 0, before reaching a plateau. Another interesting finding is that SinGAN tends to dilute (i.e. suppress the details) of semi-uniform regions in the images, which explains the observed plateau in the previous observation. As a global conclusion we can say that when used at scale 0, SinGAN displays a satisfying level of originality although some copied patches are always present.

References

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[2] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2014.

[3] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. P. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-realistic single image super-resolution using a generative adversarial network," *CoRR*, vol. abs/1609.04802, 2016. [Online]. Available: http://arxiv.org/abs/1609.04802

[4] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros, "Context encoders: Feature learning by inpainting," 2016.

[5] A. Shocher, S. Bagon, P. Isola, and M. Irani, "Internal distribution matching for natural image retargeting," *CoRR*, vol. abs/1812.00231, 2018. [Online]. Available: http://arxiv.org/abs/1812.00231

[6] T. R. Shaham, T. Dekel, and T. Michaeli, "Singan: Learning a generative model from a single natural image," *CoRR*, vol. abs/1905.01164, 2019. [Online]. Available: http://arxiv.org/abs/1905.01164

[7] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, G. Klambauer, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a nash equilibrium," *CoRR*, vol. abs/1706.08500, 2017. [Online]. Available: http://arxiv.org/abs/1706.08500

[8] D. Cozzolino, G. Poggi, and L. Verdoliva, "Efficient dense-field copy–move forgery detection," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 11, pp. 2284–2297, 2015.

[9] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman, "PatchMatch: A randomized correspondence algorithm for structural image editing," *ACM Transactions on Graphics (Proc. SIGGRAPH)*, vol. 28, no. 3, Aug. 2009.

[10] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," 2018.

[11] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved training of wasserstein gans," 2017.

[12] C. Barnes, E. Shechtman, D. B. Goldman, and A. Finkelstein, "The generalized PatchMatch correspondence algorithm," in *European Conference on Computer Vision*, Sep. 2010.

[13] D. Cozzolino, G. Poggi, and L. Verdoliva, "Copy-move forgery detection based on patchmatch," pp. 5312–5316, 2014.

[14] T. Ehret, "Automatic Detection of Internal Copy-Move Forgeries in Images," *Image Processing On Line*, vol. 8, pp. 167–191, 2018, https://doi.org/10.5201/ipol.2018.213.