



---

# IMA 206

*Synthèse de textures par automates cellulaires*

---

## Étudiants :

Mathis DUPONT  
Augustin GAVOIS  
Benjamin TERNOT

## Professeur encadrant :

Y. GOUSSEAU

# Table des matières

<b>I Introduction à l'article .....</b>	<b>2</b>
I.1 Enjeux.....	2
I.2 Intérêt des réseaux de neurones pour générer des textures .....	2
I.3 Exemples de générations.....	2
<b>II Architecture du modèle .....</b>	<b>3</b>
II.1 Automate cellulaire et système d'équations différentielles .....	3
II.2 Réseau de neurones convolutif .....	4
II.3 Canaux supplémentaires pour l'image .....	4
II.4 Entrainement du réseau de neurones .....	5
II.5 Fonction de coût .....	5
<b>III Étude des paramètres .....</b>	<b>7</b>
III.1 Ajout de fonction de coût.....	7
III.1.1 Pourquoi les matrices de Gram ? .....	7
III.1.2 Histogrammes .....	8
III.1.3 K-means revisités .....	9
III.2 Nombre de neurones .....	14
III.3 Choix des couches de <i>VGG16</i> .....	15
III.4 Nombre de canaux.....	16
III.5 Learning rate .....	17
III.6 Filtres utilisés. ....	17
III.6.1 Description des filtres .....	17
III.6.2 Filtre de Canny .....	18
III.6.3 Tests d'ablation sur les filtres.....	18
<b>Conclusion .....</b>	<b>21</b>
<b>Bibliographie.....</b>	<b>22</b>

# I Introduction à l'article

## I.1 Enjeux

Dans le domaine de la génération de texture, l'intégration de techniques d'apprentissage automatique est devenue de plus en plus courante pour créer des résultats réalistes et hautement expressifs. L'article de recherche "Texture Generation with Neural Cellular Automata" propose une approche novatrice utilisant des automates cellulaires neuronaux pour générer des textures riches et variées. Cette étude s'appuie sur les principes des automates cellulaires traditionnels et les combine avec des réseaux de neurones, afin d'améliorer les résultats existants, que ce soit en terme d'efficacité ou de rendu final.

Ce rapport vise donc à expliquer, dans un premier temps, l'architecture de cette nouvelle méthode de génération de texture, puis d'apporter des précisions sur le choix des hyper paramètres sélectionnés par les auteurs afin d'obtenir les meilleurs résultats, ainsi qu'une étude d'ablation pour mieux comprendre l'impact de chaque choix d'implémentation.

## I.2 Intérêt des réseaux de neurones pour générer des textures

Un intérêt majeur d'un réseau de neurones est qu'il peut générer de nombreuses représentations différentes d'une même texture de manière très peu coûteuse. Ainsi, on ne risque pas de reconnaître une éventuelle image dupliquée format une texture plus grande, car tout est généré avec une part d'aléatoire, en suivant les paramètres appris par le réseau.

Certes, le processus d'entraînement est coûteux en ressources et en temps, mais pour chaque texture, un seul entraînement suffit à générer d'éventuels milliers de représentations.

## I.3 Exemples de générations

La figure 1 montre quelques exemples de textures générées.

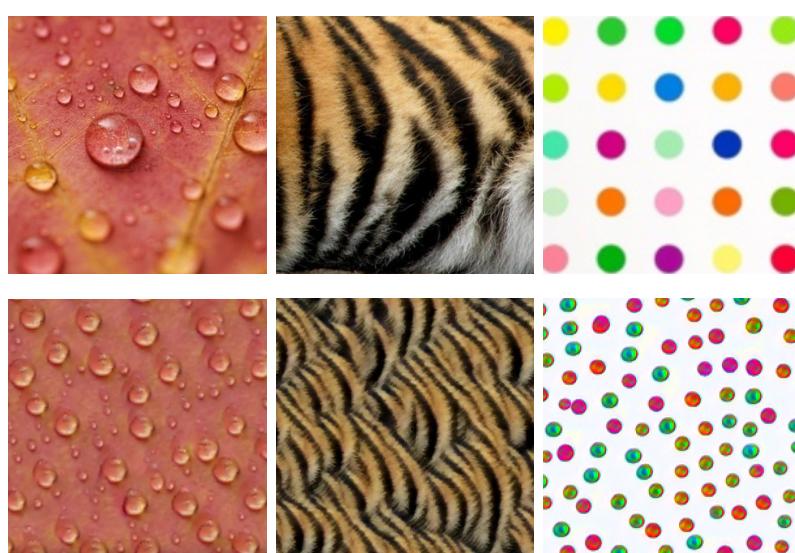


FIGURE 1 – Exemples de textures générées par l'algorithme de l'article

## II Architecture du modèle

### II.1 Automate cellulaire et système d'équations différentielles

Le but de la synthèse de texture aléatoire est de reproduire des images semblant provenir d'un processus stochastique naturel. Dans la nature, de nombreuses textures résultent d'interactions locales entre de petites particules, formant alors de plus larges structures. Ce type de modèle peut être décrit relativement simplement avec un système d'équations différentielles. Après quelques manipulations sur les équations modélisant les différents phénomènes, on obtient une équation de type équation de diffusion, qui régit l'évolution spatio-temporelle des cellules. Dans notre cas, chaque cellule représente un pixel et évolue selon la même équation que toutes les autres.

Notons  $\mathbf{s}(x, t)$  une cellule de l'image. L'équation générale d'évolution est alors :

$$\frac{\partial \mathbf{s}}{\partial t} = f(\mathbf{s}, \nabla_x \mathbf{s}, \nabla_x^2 \mathbf{s}) \quad (1)$$

Cette équation est ensuite discrétisée et intégrée, de sorte que les nouvelles images soient calculées avec la formule suivante :

$$\mathbf{s}_{t+1} = \mathbf{s}_t + f(\mathbf{p}_t) * \delta_{x,y,t} \quad (2)$$

Où  $\mathbf{p}_t = concat(\mathbf{s}, K_x * \mathbf{s}, K_y * \mathbf{s}, K_{lap} * \mathbf{s})$  est appelé *vecteur de perception* à l'instant  $t$ .  $K_x, K_y, K_{lap}$  étant les noyaux de convolution fixés  $3 \times 3$  correspondant respectivement aux filtres dérivateurs de Sobel vertical et horizontal et au filtre laplacien. Ainsi, chaque cellule dépend uniquement de ses voisins directs.  $\delta_{x,y,t}$  est l'opérateur d'échantillonage ayant une probabilité de 0,5.

C'est cette fonction  $f$  qui contient toutes les informations utiles pour générer une unique texture. Ces paramètres ne sont pas fixés à l'avance, mais sont appris par un réseau de neurones. Ce réseau est expliqué dans la section II.2.

Puisque toutes les cellules sont soumises à la même règle d'évolution, les résultats obtenus par l'automate cellulaire devraient présenter de nombreuses symétries si l'initialisation est une image uniforme. Pour conserver cette initialisation simple, les auteurs ont introduit une évolution stochastique, contrôlée par la probabilité de mise à jour de chaque cellule à chaque itération. Cela se fait lors de la multiplication avec  $\delta_{x,y,t} \in \{0, 1\}$ , où  $\mathbb{P}(\delta_{x,y,t}) = cst$ . Ici, la valeur 0,5 a été retenue. On effectue donc un tirage pour chaque cellule, et à chaque itération, puis on met à jour les images selon la formule (2). Cela revient à introduire des perturbations locales dans l'image. En effet, les cellules étant soumises à la même équation, si l'image de départ est uniforme, alors les images produites ne pourront être qu'uniforme également.

## II.2 Réseau de neurones convolutif

Comme expliqué dans la section II.1, l'apprentissage des paramètres de la fonction  $f$  se fait à l'aide d'un réseau de neurones. L'architecture retenue est un réseau convolutif récurrent. Les étages de convolution ne sont pas appris, mais fixés initialement puisqu'il s'agit des filtres déivateurs de Sobel et du filtre laplacien. La sortie de l'étage de convolution est ensuite concaténée, selon l'équation d'évolution du modèle. Les paramètres de la fonction  $f$  précédemment mentionnée sont modélisés par deux étages convolutionnels avec des noyaux de taille  $1 \times 1$  et une fonction d'activation *ReLU* entre les deux.

Un hyper-paramètre important de ce réseau de neurones est le nombre de canaux supplémentaires que l'on ajoute avant d'effectuer les convolutions fixées.

Intuitivement, si l'on gardait uniquement les trois premiers canaux RGB, on aurait un entraînement du réseau de neurones qui se fonderait intégralement sur les corrélations entre couleurs des pixels voisins. Il n'y aurait aucun moyen d'identifier dans l'image une sorte de cohérence en termes de formes de la texture retenue.

La partie III.4 (page 16) étudie l'impact du nombre de canaux retenus sur le résultat.

Finalement, l'architecture globale est relativement simple et en particulier le réseau, qui ne compte que 6 000 paramètres. Toutefois, il ne faut surtout pas négliger l'utilisation du réseau *VGG16* dans le calcul de la fonction de coût, et sans qui l'automate ne pourrait pas fonctionner.

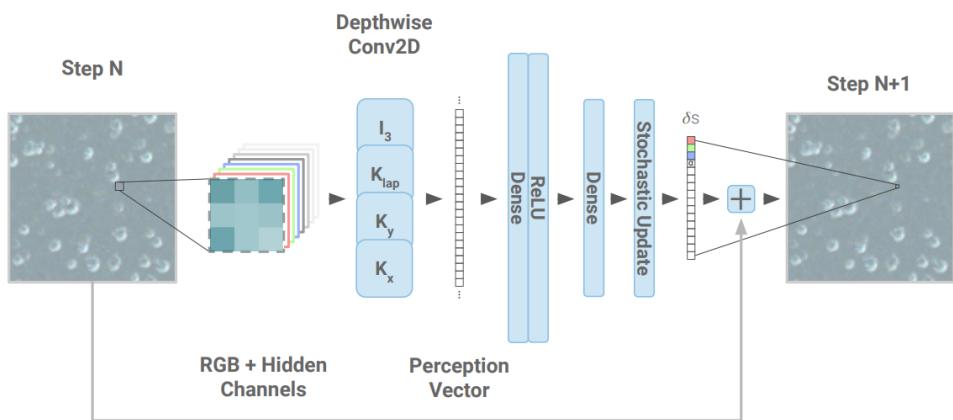


FIGURE 2 – Architecture complète

## II.3 Canaux supplémentaires pour l'image

Afin d'augmenter les dépendances entre les différents canaux RGB, les auteurs ont augmenté le nombre total de canaux. Les informations contenues dans ces canaux ne sont pas évidentes à interpréter. La couche convective  $1 \times 1$  permet de concentrer et mélanger les différentes informations générées par les filtres de convolution lors du calcul du vecteur de perception. L'effet de ces canaux est pourtant bien réel et décrit dans la section III.4

## II.4 Entrainement du réseau de neurones

L'entraînement du réseau de neurones se fait de la façon suivante. Un lot de 256 images uniformes sur les trois canaux RGB est généré. Les poids du réseau sont initialisés aléatoirement selon une loi gaussienne.

Puis pour chaque *epoch* :

- tirage de 4 images au sort dans le lot initial
- entre 32 et 64 passages successifs dans le réseau de neurones
- calcul de la fonction de coût et descente de gradient
- réinsertion des images obtenues dans le lot

L'*optimizer* utilisé est Adam, et le *learning rate* est décroissant avec les itérations, avec une valeur qui décroît de  $2 \cdot 10^{-3}$  jusqu'à  $2 \cdot 10^{-4}$ .

## II.5 Fonction de coût

La fonction de coût est une partie très importante de ce papier de recherche. En effet, son objectif est de capturer les informations de texture à plusieurs échelles différentes. Cela comprend des formes simples, rayures, mais aussi des couleurs et des structures plus larges et plus complexes. Un autre critère très important est la non-localisation des *features* calculées, puisqu'il ne faudrait pas reproduire l'image cible, mais bien la texture qu'elle représente. La position des éléments ne fait pas partie de ce qui caractérise une texture.

Il existe de nombreuses méthodes pour calculer des fonctions de pertes sur les textures. Les auteurs ont choisi d'utiliser le réseau *VGG16* pour calculer des *features* sur les images produites par le réseau et ensuite les comparer avec les *features* calculées sur l'image de texture cible.

Pour se séparer de la position de ces *features*, les auteurs utilisent les matrices de Gram calculées à partir des *features* obtenues sur les images produites. Les matrices de Gram mettent en valeur les corrélations entre les *features* plutôt que leurs positions, ce qui a exactement l'effet recherché.

Afin de capturer des informations de textures de différents niveaux, on extrait les *features* obtenus grâce aux couches convolutives de *VGG16* toutes les 5 ou 6 couches. On peut ensuite calculer simplement l'erreur quadratique moyenne (*MSE*) entre les matrices produites par le réseau et les matrices cible.

Le choix des couches n'est pas explicité dans l'article, mais apparaît dans l'implémentation proposée. Les couches 1, 6, 11, 18 et 25 ont donc été sélectionnées, probablement de façon empirique. À noter que ces couches correspondent à des activations *ReLU*, permettant d'extraire des non-linéarités dans le modèle et ainsi capturer des représentations plus discriminantes et plus complexes de notre image. Aussi, la fonction de perte implémentée par les chercheurs n'est en réalité pas celle utilisant les matrices de Gram, comme décrite dans le papier. La fonction de transport optimale a été choisie par les auteurs. Tout comme la *MSE* calculée sur les matrices de Gram, la fonction de transport optimal ne prend pas en compte la position des *features*,

puisque l'elle compare les histogrammes des *features*. Toutes choses égales par ailleurs, les résultats ne sont pas significativement différents entre les deux méthodes.

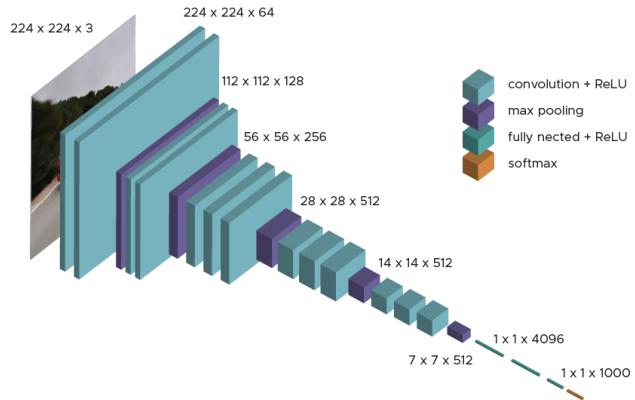


FIGURE 3 – VGG Architecture

### III Étude des paramètres

Pour toute cette partie d'étude des paramètres et de test, sauf mention contraire, la texture recherchée et donc donnée en entraînement provient de l'image de la figure 4.

Elle a l'avantage de contenir des parties très contrastées, et des formes de taille et couleurs différentes. De mauvais résultats sont ainsi rapidement détectables. À titre comparatif, les résultats donnés par les paramètres retenus de l'article sont montrés en rouge.

Afin de correctement identifier quel paramètre est responsable de quel comportement sur la génération de la texture, lorsqu'un des paramètres est testé, les autres restent fixés à la valeur retenue par les auteurs de l'article.

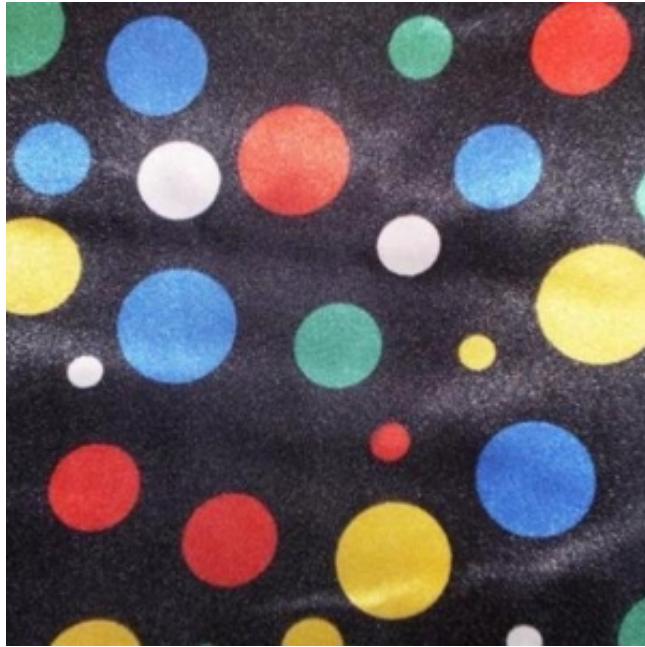


FIGURE 4 – Texture donnée pour tests

#### III.1 Ajout de fonction de coût

Nous avons pu remarquer que les fonctions de perte utilisant les *features* de *VGG16* (que ce soit le transport optimal ou les matrices de Gram) avaient du mal à bien faire respecter les couleurs de la texture cible (surtout pour l'image avec les bulles de couleurs, qui sont particulièrement saturées). On peut voir sur les figures de la partie III.2, que les deux couleurs qui reviennent toujours sont le bleu et l'orange, mixe des couleurs initiales.

##### III.1.1 Pourquoi les matrices de Gram ?

Formellement, une matrice de Gram est définie par

$$G_{i,j}(F) = \sum_k F_{ik} F_{kj} \quad (3)$$

Où  $F_{ij}$  désigne correspond au  $j$ -ème pixel de la  $i$ -ème *feature map*. Elle met en lumière les corrélations qui peuvent exister entre les features sans prendre en compte la position spatiale

de celles-ci. C'est donc un outil idéal pour la caractérisation de texture. Cette méthode possède pourtant des limites, dont la principale est l'invariance par changement de moyenne des *features*. Cela peut entraîner des instabilités voire des divergences de la *loss*. Nous avons donc eu l'idée de palier ce problème en ajoutant simplement un terme MSE sur les moyennes et variances des *features* produites par *VGG16*.

### III.1.2 Histogrammes

L'article de recherche [6] soulève le problème suivant : deux matrices de Gram identiques peuvent provenir de distribution différentes en terme de moyenne et de variance. Les auteurs suggèrent alors de recourir à des statistiques de plusieurs ordres différents pour améliorer la stabilité des résultats. Leur méthode consiste à faire correspondre les histogrammes des différentes activations de *VGG16* des images sources aux histogrammes des activations de l'image cible. Finalement, une loss MSE est utilisée pour comparer les activations re-distribuées et les activations cibles. Cela permet de garantir une distribution statistiques la plus proche possible entre les images produites par le NCA et l'image cible.

Notre implémentation n'a malheureusement pas fonctionné à temps pour pouvoir montrer des résultats probants. Voici tout de même le code écrit qui permettra de comprendre l'architecture de la *loss*.

```
1
2 def histogram_matching(source, target):
3     '''This function remaps the histograms of the features activations
4         of the source tensor to match those of the target tensor. '''
5     remap_sources = []
6     for i in range(len(source)):
7         b, c, _ = source[i].shape
8         source_2d = source[i].view(b*c, -1)
9         target_2d = target[i].view(c, -1).repeat(b, 1)
10
11         source_cdf = torch.histc(source_2d, bins=256, min=0, max=255).
12             cumsum(dim=0)
13         target_cdf = torch.histc(target_2d, bins=256, min=0, max=255).
14             cumsum(dim=0)
15
16         source_cdf_normalized = source_cdf / source_cdf.sum(dim=0)
17         target_cdf_normalized = target_cdf / target_cdf.sum(dim=0)
18
19         mapping = torch.zeros(256, dtype=torch.float32)
20         for p in range(256):
21             source_val = source_cdf_normalized[p]
22             min_diff = float('inf')
23             min_idx = 0
24             for j in range(256):
25                 target_val = target_cdf_normalized[j]
26                 diff = torch.abs(source_val - target_val)
27                 if diff < min_diff:
```

```

25         min_diff = diff
26         min_idx = j
27         mapping[p] = min_idx
28
29     # Apply the mapping to the source tensor
30     remap_tensor = torch.zeros_like(source[i])
31     for bi in range(b):
32         for ci in range(c):
33             remap_tensor[bi, ci] = mapping[source[i][bi, ci].long()].
34             clamp(0, 255)
35     remap_sources.append(remap_tensor)
36
37
38 def create_hist_loss(target_img):
39     """ This function creates the mse loss function between the
40         remapped features activations and the target activations"""
41     yy = calc_styles_vgg(target_img)
42     def loss_f(imgs):
43         xx = calc_styles_vgg(imgs)
44         xx_remap = hist_match(xx, yy)
45         return sum(mse_loss(x, y) for x, y in zip(xx_remap, yy))
46     return loss_f

```

### III.1.3 K-means revisités

Pour tenter de faire respecter à l'image générée les couleurs de l'image cible, nous avons également pensé à appliquer l'algorithme des kmeans, non pas sur la distance spatiale entre deux pixels, mais sur une distance de couleur. Cependant le calcul des kmeans sur une image n'est pas une opération différentiable, notamment dû à la fonction argmin pour l'assignation des pixels aux différents clusters. Nous nous sommes donc inspiré des kmeans pour créer une fonction de coût différentiable.

```

46 def create_kmeans_loss(target_img, num_cluster):
47     X = target_img.view(3, 16384).transpose(0,1)
48     cluster_ids_x, cluster_centers = kmeans(X=X, num_clusters=
49         num_cluster, distance='euclidean', device=device) # cf https
50         ://pypi.org/project/kmeans-pytorch/
51     pixel_per_cluster = torch.bincount(cluster_ids_x, minlength=
52         num_cluster)
53     print(pixel_per_cluster)
54     fig, ax = plt.subplots(1,num_cluster, figsize=(10,10))
55     for i in range(num_cluster):
56         img = np.full((10, 10, 3), cluster_centers[i].numpy())
57         ax[i].imshow(img)
58         ax[i].set_title(f"nb_pixel={pixel_per_cluster[i]}", fontsize=6)

```

```

56     ax[i].axis('off')
57     plt.show()
58     cluster_centers_extended = cluster_centers.unsqueeze(0).expand
59         (16384, -1, -1)
60     def err(imgs):
61         error_distance = torch.tensor(float(0), requires_grad=True)
62         error_distribution = torch.tensor(float(0), requires_grad=True)
63         for i in range(imgs.shape[0]):
64             Y = torch.clamp(imgs[i], 0, 1).view(3, 16384).transpose(0,1)
65             mapping_kmeans = cluster_centers_extended.to(device) - Y[:, None, :].to(device)
66             squared_distances = torch.sum(mapping_kmeans.pow(2), dim=2)
67             temperature = 0.5
68             probs = torch.nn.functional.softmax(-squared_distances /
69                 temperature, dim=1)
70             prob_per_cluster = torch.sum(probs, dim=0)
71             diff_per_cluster = torch.abs(prob_per_cluster.to(device) -
72                 pixel_per_cluster.to(device))
73             distances_normalized = probs*squared_distances
74             error_distance = error_distance + distances_normalized.sum()
75             error_distribution = error_distribution + torch.sqrt(torch.
76                 sum(diff_per_cluster**2))
77         return error_distance, error_distribution, diff_per_cluster
78     return err

```

Dans un premier temps, nous calculons les *kmeans* sur l'image cible une bonne fois pour toute, et nous stockons le nombre de pixels par cluster ainsi que les centres de chacun.

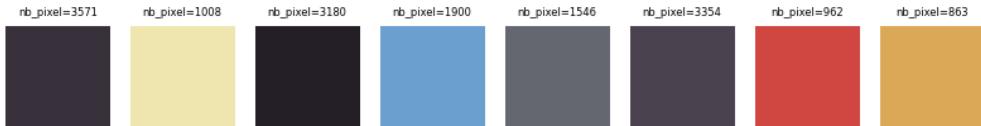


FIGURE 5 – K-Mean image cible

Ensuite pour chaque image que l'on donne en entrée, nous allons calculer la distance entre chacun de ses pixels et les différents centres des clusters. Cette distance est donnée par *mapping\_kmeans*. Seulement au lieu de choisir avec un *argmin* le cluster le plus proche pour chaque pixel, nous allons utiliser la fonction *torch.nn.functional.softmax*. Cette fonction est bien différentiable, et nous renvoie une probabilité pour ce pixel d'appartenir à chacun des clusters. Nous discuterons après du paramètre de température qui permet soit de rendre *softmax* très proche de *argmin*, soit de rendre les probabilités assez uniformes si par exemple

un pixel est aussi loin d'un cluster que de l'autre.

Grâce à ce softmax, on peut calculer deux fonctions de coût qui seront pondéré par ces probabilités :

- error\_distance qui correspond à l'écart entre chaque pixel et les différents clusters, pondéré par le softmax
- error\_distribution qui correspond à l'écart entre le nombre de pixels dans chaque cluster de l'image cible, et le nombre de pixels (pondéré par le softmax) qui sont rattachés aux différents clusters de l'image cible.

Les deux erreurs tournent autout de  $1 \cdot 10^4$ , or l'erreur liée aux features *VGG* est de l'ordre de  $1 \cdot 10^8 - 1 \cdot 10^9$ . C'est donc pour cela qu'il faut multiplier par un coefficient constant nos deux loss dans la training loop. Il ne faut pas que ce coefficient soit trop grand sous peine de ne plus utiliser la loss *VGG* qui est la principale, mais si ce coefficient est trop petit les *kmeans* n'auront pas d'impact sur l'image générée.

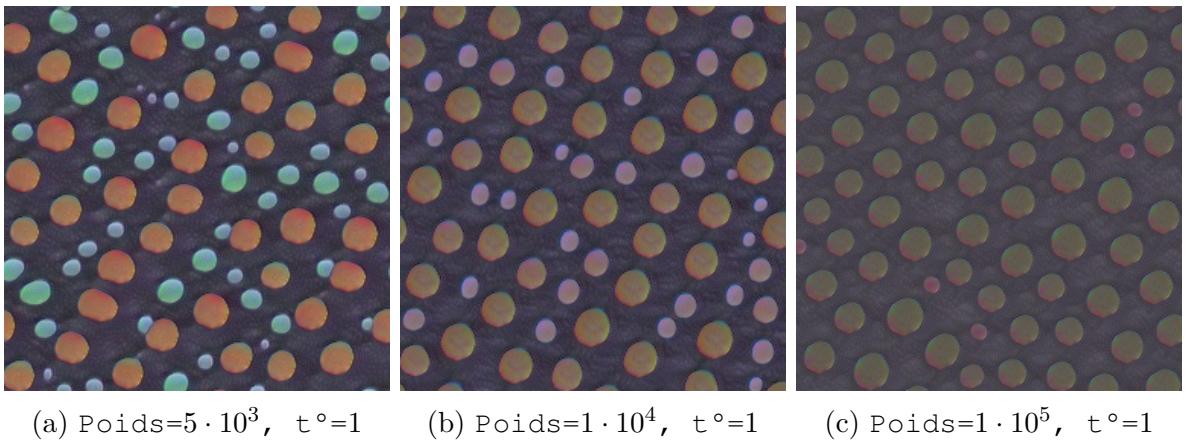
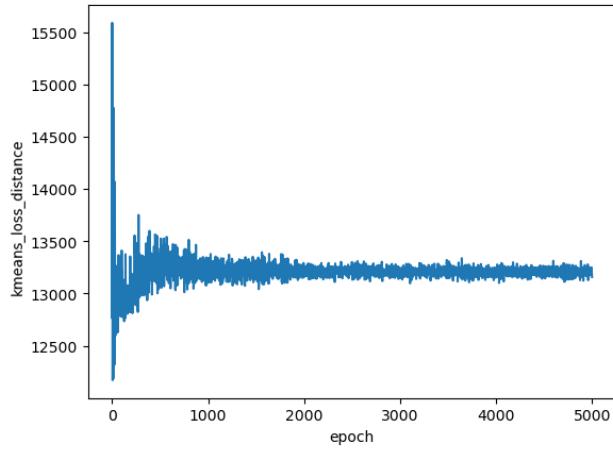


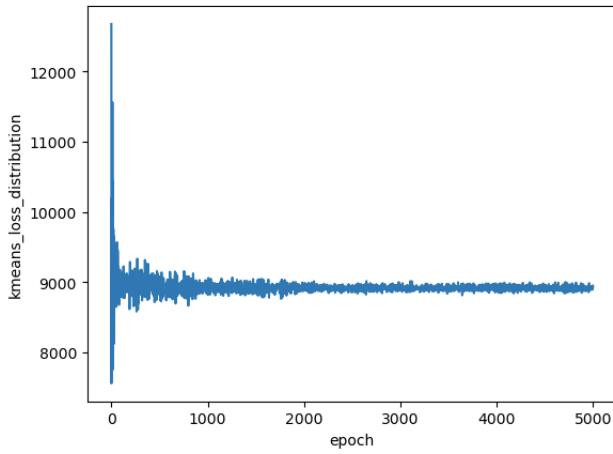
FIGURE 6 – Modification des poids

Nous observons sur la figure 6 que plus le poids de la distance et de la distribution des kmeans augmente, plus l'image devient sombre et il n'y a pas beaucoup de variation de couleur. Si l'on se réfère à la figure 7a et 7b, on peut voir que le gradient est très rapidement "satisfait" et il ne bouge plus beaucoup après.

On peut observer que avec une température de 1, la *training loop* a du mal à apprendre sur la distance et la distribution, car beaucoup de pixels sont catégorisés "noirs" et restent bloqués. Regardons l'impact de la température.

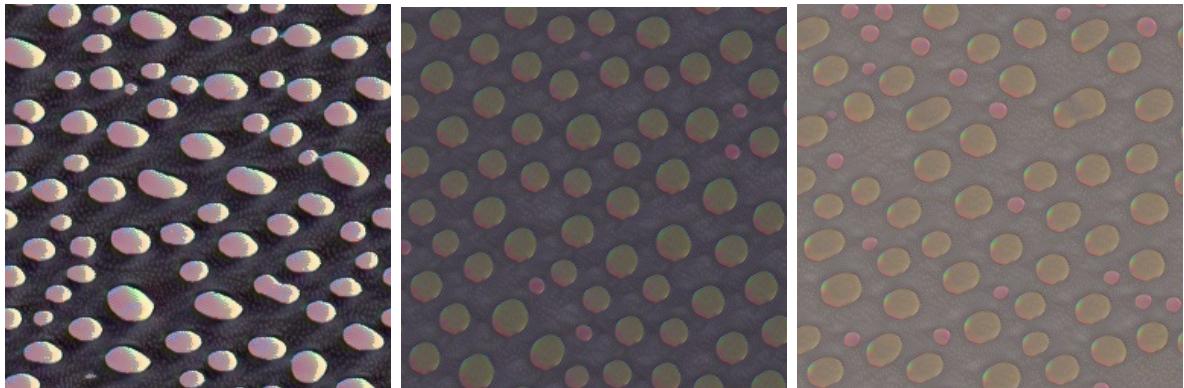


(a) Distance : Poids= $1 \cdot 10^5$ ,  $\tau^\circ=1$



(b) Distribution : Poids= $1 \cdot 10^5$ ,  $\tau^\circ=1$

FIGURE 7 – Distance et distribution pour Poids= $1 \cdot 10^4$ ,  $\tau^\circ=1$



(a) Poids= $1 \cdot 10^5$ ,  $\tau^\circ=0.1$       (b) Poids= $1 \cdot 10^5$ ,  $\tau^\circ=1$       (c) Poids= $1 \cdot 10^5$ ,  $\tau^\circ=10$

FIGURE 8 – Modification de la température

On a :

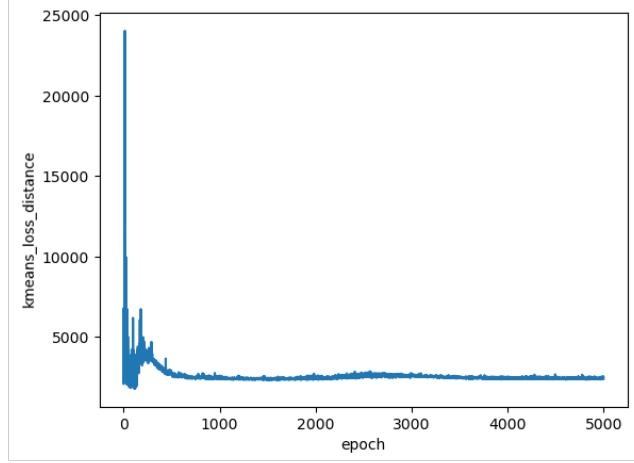
$$\text{softmax}(x)_i = \frac{\exp \frac{y_i}{T}}{\sum_j \exp \frac{y_j}{T}}$$

Ainsi, si l'on a  $T = 10$ , on va lisser les probabilités et ainsi obtenir une probabilité assez uniforme pour l'assignation des pixels aux clusters. Cela se reflète par des couleurs moyennes :

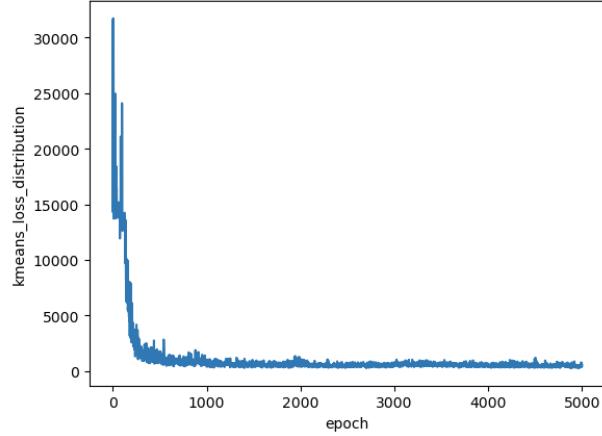
l'algorithme va essayer de trouver une couleur qui satisfait tout le monde (cf. 8c).

Avec  $T = 0.1$ , on voit qu'on se rapproche plus d'un *argmin*, et on privilégie ainsi un cluster par rapport aux autres. Cela va permettre notamment aux points noirs d'être noirs. Cette valeur de la température a un meilleur impact sur l'image générée si l'on regarde l'image 8a.

Au vu également des courbes 9a et 9b, on voit que la *backpropagation* arrive mieux à apprendre sur les *kmeans*, notamment sur la distribution.



(a) Distance : Poids= $1 \cdot 10^5$ ,  $t^\circ=1$



(b) Distribution : Poids= $1 \cdot 10^5$ ,  $t^\circ=1$

Afin de diversifier les types d'images générées, la figure 10 montre d'autres générations d'image, avec comme paramètres : poids =  $1 \cdot 10^4$ ,  $t^\circ = 0.1$ .

En conclusion, on peut dire que ces *kmeans* revisités peuvent avoir une influence sur les couleurs de l'image si les hyper-paramètres sont bien choisis. Pour les images avec très peu de nuance de couleur (Hexagones par exemple), on peut tout de même observer que l'algorithme converge plus rapidement vers les bonnes couleurs, et peut donc plus rapidement apprendre sur les caractéristiques plus "spécifiques" à l'image.

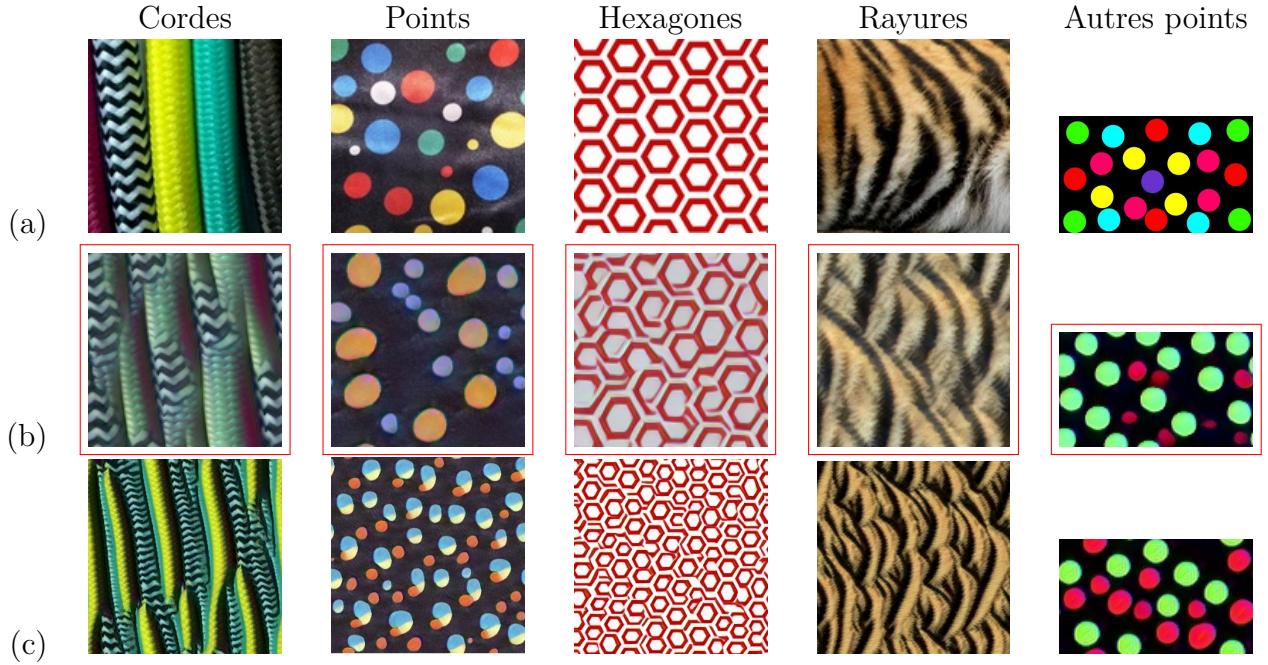


FIGURE 10 – Génération d’image avec les *kmeans* :  
(a) Texture cible, (b) Param par défaut, (c) Kmeans

### III.2 Nombre de neurones

Le réseau de neurones étant constitué de 2 couches, le seul paramètre variable concernant le nombre de neurones est le nombre de neurones entre la couche 1 et la couche 2. Le code provenant de l’article en utilise *hidden\_n* = 96.

Afin d’analyser l’impact de ce nombre sur les résultats obtenus, nous avons essayé avec 4, 8, 16, 32, 64, 128 et 256 neurones, pour comparer avec le résultat de l’article (cf figure 11).

Qualitativement, on voit que la qualité des résultats augmente avec le nombre de neurones du réseau. À partir de *hidden\_n* = 32, on retrouve des résultats qui ressemblent assez bien à la texture que l’on cherche à générer (en ce qui concerne la photo de taches colorées sur fond noir).

On peut toutefois noter que sur certains tests, le réseau a appris de manière non-optimale : par exemple avec *hidden\_n* = 96 (le paramètre choisi dans l’article), le réseau a appris qu’un petit rond rouge/orangé devait être adjacent à un gros rond cyan, et ainsi, sur différentes images générées, on retrouve toujours cette disposition.

Un autre exemple est celui avec *hidden\_n* = 128, pour lequel le réseau a appris sans tenir compte des ronds rouges. Notons qu’ajouter des neurones améliore *en général* le résultat, mais que le temps d’entraînement est significativement augmenté (plus d’une heure pour *hidden\_n* = 128, et presque 2 h pour *hidden\_n* = 256)

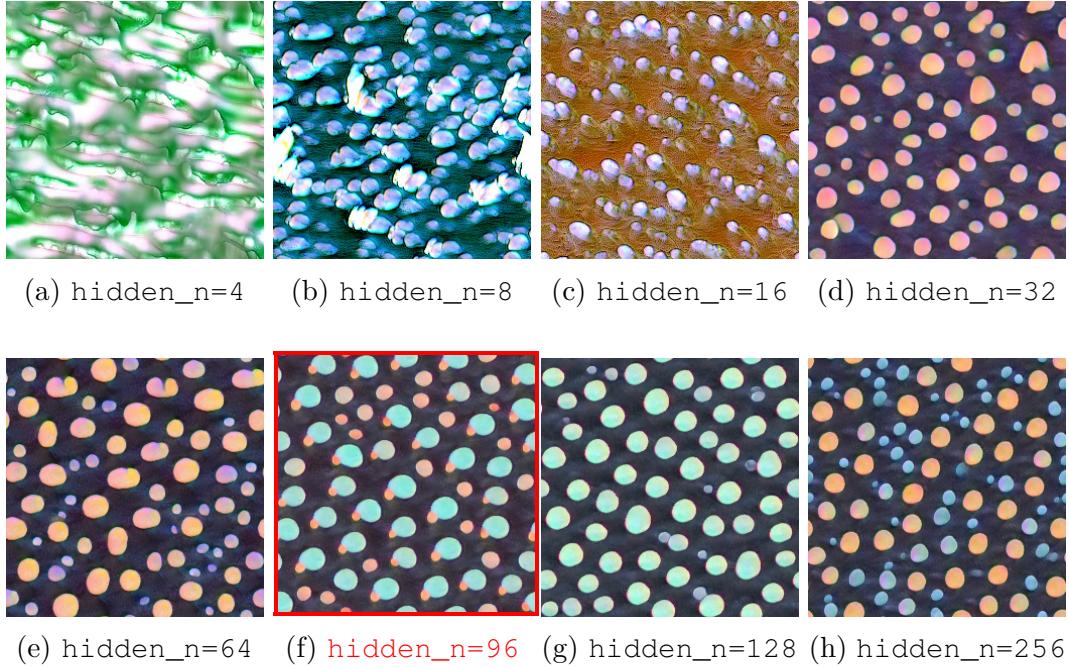


FIGURE 11 – Textures obtenues en fonction de `hidden_n`

### III.3 Choix des couches de *VGG16*

Afin de récupérer des *features* abstraites permettant de caractériser correctement des textures, on utilise un réseau très profond permettant de faire de la classification : *VGG16*. Les détails de l’architecture ne sont pas nécessaire à la compréhension de cette partie. On retiendra simplement que dans les réseaux très profonds, plus les couches sont proches de la sorties, plus elles permettent d’encoder des informations complexes et abstraites. A l’inverse, les activations issues des premières couches caractérisent des éléments géométriques relativement simple et compréhensible. Il est courant de retrouver des filtres dérivateurs ou détecteurs de contours dans les couches basses.

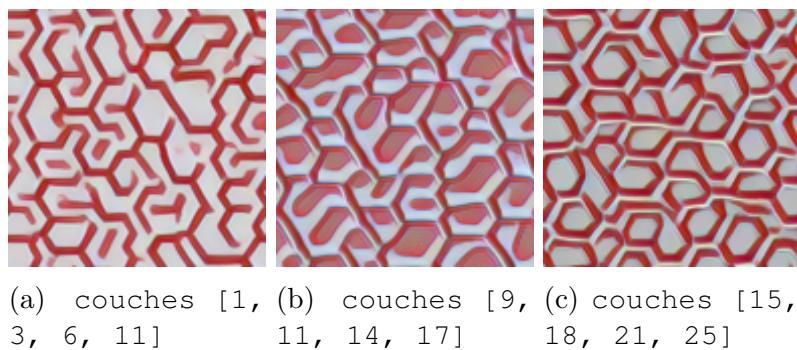


FIGURE 12 – Textures obtenues en fonction du choix des couches de VGG16

Comme on peut le voir sur la figure 12, les couches de bas niveaux force les textures générées à respecter des règles très simple géométriquement. En effet le réseau a apprit que les barres rouges devaient se déplacer dans seulement trois directions. Il a appris que le fond était blanc mais pas que les barres rouges doivent former des hexagones.

A l’inverse, les couches de haut niveaux semblent avoir mieux compris l’idée des hexagone mais il manque la règle des trois uniques directions.

Pour ce qui est des couches intermédiaires, nous n'avons pas vraiment d'interprétation. Il faudra observer d'autres textures pour avoir un effet plus remarquable.

### III.4 Nombre de canaux

Comme énoncé dans la partie II.2, des canaux supplémentaires sont ajoutés à ceux représentant l'image en RGB, afin de permettre une génération plus complexe que simplement basée sur les corrélations des couleurs de pixels voisins.

Afin de comprendre réellement l'impact de ce nombre de canaux supplémentaires, nous avons testé différentes valeurs de  $chn$  (cf figure 13).

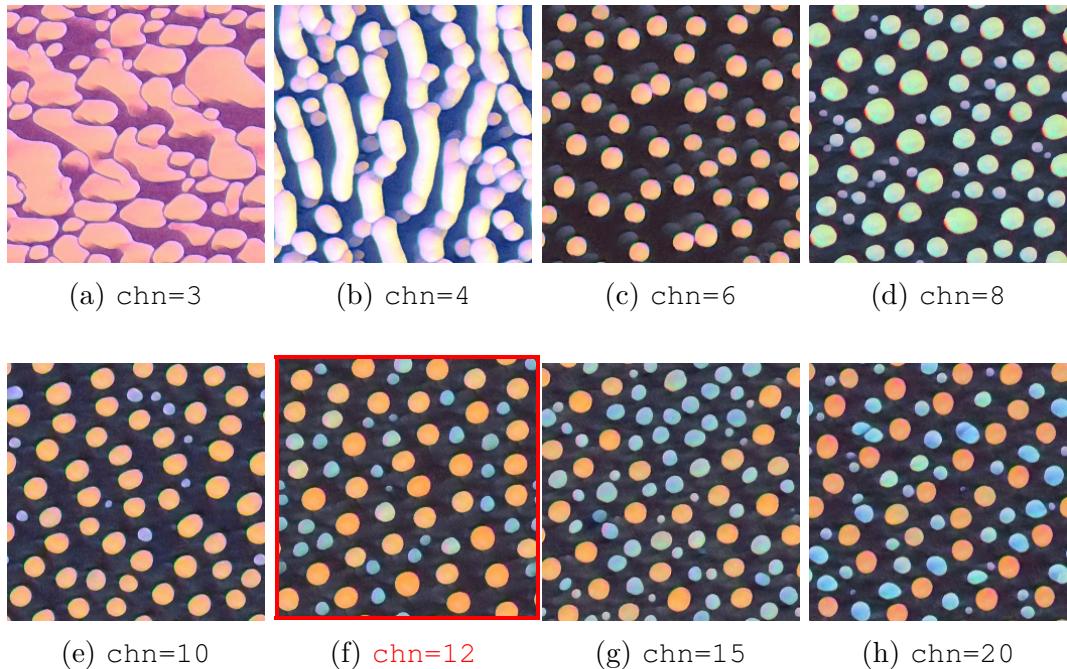


FIGURE 13 – Textures obtenues en fonction de  $chn$

Comme on pouvait s'y attendre, sans canal supplémentaire (fig (a) de la figure 13), la texture générée ne présente aucune cohérence en terme de forme, et même les couleurs sont incorrectement représentées.

Avec seulement 3 canaux supplémentaires ( $chn = 6$ ), on commence à trouver des formes reconnaissables. Cependant, avant  $chn = 12$  voire même  $chn = 15$ , la distribution de la texture et des formes de couleur n'est pas du tout représentative de la texture visée.

Ainsi, on pourrait supposer que les canaux supplémentaires permettent à la fois d'obtenir des formes cohérentes au sein de la texture, ainsi que de jouer sur leur distribution les unes par rapport aux autres.

### III.5 Learning rate

Afin de choisir le *learning rate*, on peut suivre la procédure suivante : fixer une valeur quelconque puis tracer l'évolution de la fonction de perte pour plusieurs *epochs*. Si la fonction de perte n'évolue que trop peu, alors la valeur initiale du *learning rate* est trop faible. Dans le cas où ce *learning rate* serait trop important, la fonction de perte alternerait entre des valeurs très grandes et très faibles.

Une fois la valeur du *learning rate* correctement fixée, on peut augmenter le nombre d'epochs jusqu'à ce que, soit les résultats atteignent les attentes, soit la fonction de perte recommence à alterner entre des valeurs grandes et faibles autour d'une position stagnante.

Nos expérimentations sont arrivées à la même conclusion que les auteurs, avec un *learning rate* de l'ordre de  $1 \cdot 10^{-3}$ . Une astuce d'implémentation permet de faire varier la valeur du *learning rate* au cours du temps. Cela permet de converger plus rapidement vers la solution au début de l'entraînement puis d'avancer de façon plus précise à mesure que le réseau se rapproche du minimum local.

Un outil de *PyTorch* est utilisé pour réduire le *learning rate* à mesure que l'entraînement progresse : `torch.optim.lr_scheduler.MultiStepLR(opt, [1000, 2000], 0.3)`. Cela permet, aux *epochs* 1000 et 2000 de multiplier par 0.3 le *learning rate*. Si l'on essaye de rendre le *learning rate* encore plus petit (multiplier 5 ou 6 fois par 0.3 par exemple), cela n'apporte pas grand chose, a contrario même, cela ralentit énormément le programme car il faudrait encore plus d'*epochs* pour converger.

### III.6 Filtres utilisés

#### III.6.1 Description des filtres

Comme décrite sur l'architecture du modèle (cf Figure 2), l'extraction de features pour l'entraînement de notre réseau de neurone utilise différentes matrices de filtres : l'identité, le Laplacien et le filtre de Sobel (selon l'axe x et selon l'axe y de manière indépendante).

L'utilisation de l'identité permet de récupérer les informations des pixels directement, pour s'en servir dans l'entraînement du modèle.

Le Laplacien permet d'utiliser la dérivée pour chaque canal, qui peut être source d'information sur les textures et ainsi servir à notre modèle.

Le filtre de Sobel est un filtre de taille  $3 \times 3$  qui permet de récupérer l'information des pixels formant des bords de texture, dans le sens vertical comme horizontal et complète les informations du Laplacien.

D'autres filtres ont été testés pour notre modèle : le filtre médian et le filtre de Canny (cf description partie III.6.2).

Afin de voir l'impact de chacun de ces éléments sur l'entraînement de notre modèle, nous avons procédé à des tests d'ablation, cf partie III.6.3.

### III.6.2 Filtre de Canny

Le filtre de Canny est un filtre qui, comme celui de Sobel, permet de détecter les éléments formants les bords d'objets présents dans une image. Contrairement au filtre de Sobel ou à des filtres moyenneurs, il ne se présente pas simplement sous forme de convolution, mais est un algorithme complet :

- Flou gaussien : L'objectif principal est de réduire le bruit de l'image et de prévenir les détections de contours indésirables ou erronées.
- Filtrage par le gradient : ensuite, on utilise un opérateur de gradient, comme le filtre de Sobel, pour calculer les variations d'intensité des pixels dans l'image. Ce filtre calcule les dérivées partielles horizontales et verticales de l'image, ce qui permet de mettre en évidence les régions où le changement d'intensité est le plus fort, qui correspondent généralement aux zones de bords d'objets.
- Suppression des non-maxima : une fois les gradients calculés, l'algorithme parcourt chaque pixel de l'image et vérifie s'il est un maximum local dans la direction du gradient. Cela signifie que seuls les pixels qui représentent les variations d'intensité les plus fortes dans leur direction respective sont conservés, tandis que les autres pixels sont supprimés. Cette étape permet de réduire l'épaisseur des contours et de rendre leur localisation plus précise.
- Seuil des hystérosis : À ce stade, certains pixels peuvent encore être considérés comme des contours même s'ils ne sont pas très significatifs. Pour résoudre ce problème, le filtre de Canny utilise une technique appelée seuil d'hystérosis. Cette technique utilise deux seuils : un seuil inférieur et un seuil supérieur. Les pixels ayant une valeur de gradient supérieure au seuil supérieur sont considérés comme des contours forts et sont conservés. Les pixels ayant une valeur de gradient inférieure au seuil inférieur sont considérés comme des contours faibles et sont rejettés. Les pixels ayant une valeur de gradient entre les deux seuils sont conservés uniquement s'ils sont connectés à des pixels forts. Cela permet d'éliminer les contours indésirables et de conserver uniquement les contours significatifs.

Grâce à cet algorithme plus complet que simplement le passage par filtre de Sobel, le filtre de Canny offre plusieurs avantages. Il est notamment plus précis en terme de localisation des contours, car ceux-ci sont bien moins épais que obtenus par Sobel (étape de suppression des non-maxima). Il est aussi moins sensible au bruit grâce à l'utilisation de seuils d'hystérosis, qui permettent de supprimer les contours indésirables potentiellement détectés par le filtre de Sobel.

Un désavantage majeur cependant est qu'il est beaucoup plus coûteux en terme de ressources, nous nous en sommes rendu compte après l'implémentation.

### III.6.3 Tests d'ablation sur les filtres

La figure 14 présente les résultats obtenus pour l'entraînement de 5 images selon la présence ou non de différents filtres. Malheureusement, pour des raisons pratiques, le filtre de Canny s'est avéré être inexploitable car bien trop coûteux (plus de 20 minutes par *epoch*, sachant que le total d'*epoch* recherché est de l'ordre de plusieurs milliers).

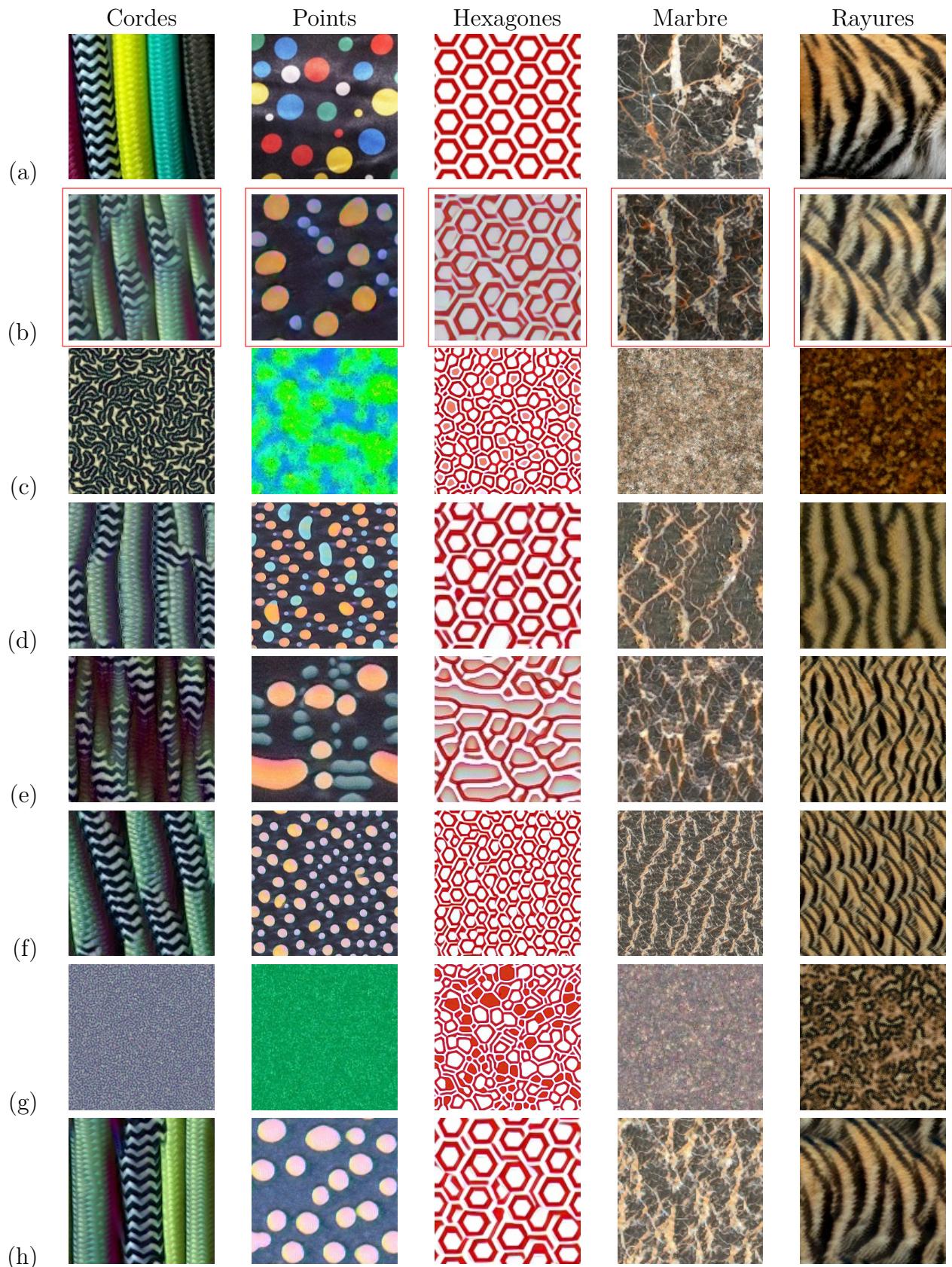


FIGURE 14 – Résultats selon les filtres utilisés :

- (a) Texture cible,
- (b) Laplace+Sobel,
- (c) Laplace,
- (d) Laplace+Sobel\_x,
- (e) Laplace+Sobel\_y,
- (f) Sobel,
- (g) Laplace+Moyen,
- (h) Laplace+Sobel+Moyen

On remarque que l'utilisation du filtre de Sobel influence grandement le résultat obtenu : si l'on omme<sup>t</sup> le filtre suivant l'un des axes, la texture générée est déformée selon cet axe, les éléments la composant pouvant être étirés dans cette direction (voir notamment la figure des points et celle des hexagones). Lorsqu'on enlève complètement ce filtre, les bords des éléments constitutifs de la textures ne sont plus détectés, et le résultat obtenu a perdu toute la cohésion qui était présente dans la texture cible.

Cependant, l'utilisation spécifique du filtre de Sobel uniquement dans un seul axe peut s'avérer utile : si l'on s'intéresse à la texture tigrée, afin d'éviter des artefacts horizontaux ou diagonaux, on peut se restreindre à l'utilisation de Sobel\_x et ainsi obtenir de belles rayures verticales.

En ce qui concerne l'utilisation du Laplacien, on remarque que l'ommettre engendre systématiquement des générations composées d'innombrables motifs plus petits que ceux présents dans la texture cible. Ainsi, le Laplacien qui donne l'information sur la dérivée et donc sur la position relative d'un pixel dans une forme est essentiel à la bonne génération.

L'utilisation d'un filtre moyen couplé au Laplacien ne permet pas de palier au manque de l'information des bords donnée par Sobel, au contraire, on peut même voir apparaître des artefacts et des zones d'aplats de couleur non présentes sur la texture cible. Par exemple, pour la texture hexagonale, certaines formes générées sans Sobel et avec le filtre moyen vont être colorées entièrement en rouge. Pour la texture tigrée, on perd toute cohérence de texture en rayure, représentative du tigre, mais on observe plus un effet tacheté.

Néanmoins, lorsque l'on couple le filtre Laplacien, celui de Sobel et le filtre moyen, nous obtenons des résultats qui semblent parfois meilleurs pour certaines textures que sans le filtre moyen (par exemple les cordes ou les rayures de tigre).

## Conclusion

En partant d'une modélisation locale d'interactions entre des cellules, les auteurs sont parvenus à établir un modèle pour générer des textures en résolvant une équation différentielle grâce à un réseau de neurones convolutif récurrent. Si l'existence des automates cellulaires d'une part et des réseaux de neurones convolutifs récurrents d'autre part était déjà actée, la réunion de ces deux modèles appliquée à la génération de texture est bien une idée novatrice.

Il est assez intéressant de constater que les interactions locales permettent de faire émerger des structures nettement plus grandes, de l'ordre de quelques dizaines de pixels. Ces structures sont d'autant plus larges que la capacité du réseau est importante.

Bien que le réseau de neurones soit relativement très compact en termes de nombre de paramètres, l'utilisation du réseau *VGG16* pour le calcul de la fonction de perte rend l'entraînement significativement plus long.

Nous avons fait de nombreux tests sur les hyper-paramètres pour mieux comprendre leurs effets sur la production finale de l'algorithme. Sans grande surprise, les valeurs choisies par les auteurs étaient déjà optimales dans de nombreux cas. Il est possible d'augmenter la capacité du réseau pour obtenir de meilleurs résultats, mais cela se fait évidemment au détriment du temps d'entraînement qui peut rapidement exploser.

Alors que nous n'étions que peu satisfaits du respect des couleurs, nous avons essayé d'implémenter des termes supplémentaires à la fonction de perte afin de pénaliser la différence entre les images cible et les images produites. Cela n'a malheureusement pas abouti à une amélioration nette des résultats.

Souvent, les articles de recherche présentent les informations de manière très compacte et peu explicative. Si comprendre les grandes lignes peut être fait en lisant l'introduction et la présentation des résultats, saisir les détails pour une compréhension plus fine relève d'une réelle compétence. Et c'est à travers ce type de projet que nous pouvons la développer.

## Bibliographie

- [1] L. A. Gatys, A. S. Ecker, and M. Bethge, *Texture Synthesis Using Convolutional Neural Networks*, Advances in Neural Information Processing Systems 28, 2015.  
Available at : <http://bethgelab.org/deptextures/>
- [2] A. Mordvintsev, E. Niklasson, and E. Randazzo, *Texture Generation with Neural Cellular Automata*, Preprint, 2021.  
Available at : <https://arxiv.org/abs/2105.07299>  
Demo at : <https://distill.pub/selforg/2021/textures/>  
Code at : <https://github.com/google-research/self-organising-systems>
- [3] K. Simonyan and A. Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014.  
Available at : <https://arxiv.org/abs/1409.1556>
- [4] *Describable Textures Dataset (DTD)*  
Available at : <https://www.robots.ox.ac.uk/~vgg/data/dtd/>
- [5] T. Gibara, *CannyEdgesDetector.java*  
Available at :  
<http://www.tomgibara.com/computer-vision/CannyEdgeDetector.java>
- [6] Eric Risser1, Pierre Wilmot and Connelly Barnes *Stable and Controllable Neural Texture Synthesis and Style Transfer Using Histogram Losses*  
Available at :  
<https://arxiv.org/pdf/1701.08893.pdf>