JULIE DIGNE

NOVEMBRE 2023

Modèles statistiques

RÉDIGÉ PAR

SPATARO Mathis - p1819506
ROULLIER Léa - p1911736

SYNTHÈSE DE TEXTURES

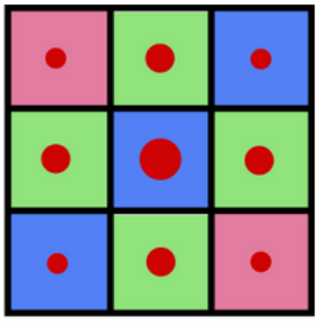
Nom du fichier correspondant au code : texture_synthesis.py

Le code génère une ***texture synthétique en utilisant des patches*** de l'image d'exemple, en se basant sur des ***critères de similarité*** pondérés et en utilisant un ***masque gaussien*** pour contrôler l'influence des pixels voisins. Le processus de génération se fait de manière ***itérative***, en remplissant les pixels de la texture synthétique en fonction de la ***similarité avec ces patches***.

COMPOSANTS DU CODE

Masque Gaussien 2D

La fonction ***normalized_gaussian_2d_mask*** génère un masque gaussien 2D normalisé. Ce masque permet de ***pondérer les différences entre les patches*** lors de la synthèse de texture. La normalisation garantit que la ***somme de toutes les valeurs du masque est égale à 1***, ce qui est essentiel pour une pondération correcte.



Représentation visuelle de la gaussienne 2D et de la manière dont elle affecte le résultat : la taille du rond est proportionnelle au coefficient associé à chaque pixel autour du centre.

EXEMPLE

```
patch1 = np.array([[100, 120, 90],
                   [80, 110, 130],
                   [70, 100, 110]])

patch2 = np.array([[110, 130, 80],
                   [90, 100, 120],
                   [120, 110, 100]])
```

Patches d'exemples

```
weight_mask = np.array([[0.1, 0.2, 0.1],
                        [0.2, 0.4, 0.2],
                        [0.1, 0.2, 0.1]])
```

Masque de poids gaussien générique (non normalisé)

Somme des Différences au Carré Pondérées

La fonction ***weighted_ssd*** mesure la ***somme des différences au carré pondérées entre deux patches d'images***. Elle prend en compte le ***masque gaussien*** pour pondérer les différences et normalise le résultat. La somme est effectuée sur les ***canaux de couleur***, et le résultat final est normalisé par la somme du masque de poids.

SUITE EXEMPLE

```
squared_differences = [
    [100, 100, 100],
    [100, 0, 100],
    [2500, 0, 1210]
]

weighted_ssd_result = (
    0.1 * 100 + 0.2 * 100 + 0.1 * 100 +
    0.2 * 0 + 0.4 * 100 + 0.2 * 0 +          = 470
    0.1 * 2500 + 0.2 * 0 + 0.1 * 1210
)

normalized_weighted_ssd_result = weighted_ssd_result / (
    0.1 + 0.2 + 0.1 + 0.2 + 0.4 + 0.2 + 0.1 + 0.2 + 0.1
)                                 = 47
```

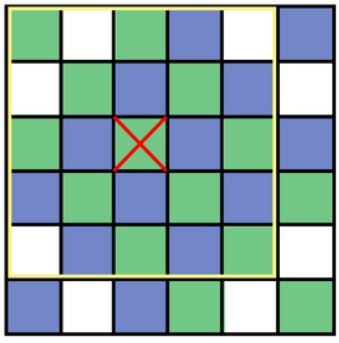
(patch1 - patch2) ** 2

np.sum(squared_differences * weight_mask)

weighted_ssd_result / np.sum(weight_mask)

Extraction de patch

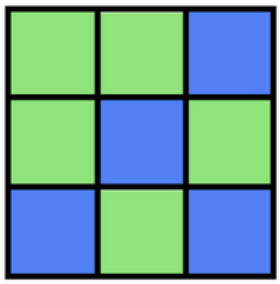
La fonction ***get_patch_from_image*** permet d'***extraitre un patch*** d'une image en fonction d'un centre donné, d'une taille de fenêtre spécifiée et d'une valeur par défaut pour les pixels en dehors de l'image. Elle garantit que le patch ne dépasse pas les limites de l'image.



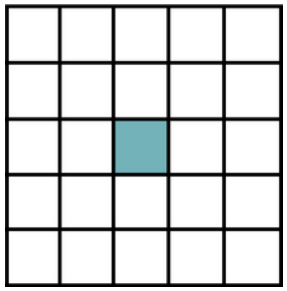
Extrait d'un patch 5x5 (carré jaune) à partir de son pixel central (croix rouge).

Classe TextureGenerator

La classe ***TextureGenerator encapsule le processus de synthèse de texture***. Elle ***initialise les attributs*** nécessaires tels que la taille de la fenêtre, la taille de la texture de sortie, et charge l'image d'exemple. Elle ***extrait les patches*** de l'image d'exemple et ***initialise l'image synthétique avec le pixel central***.



Patch



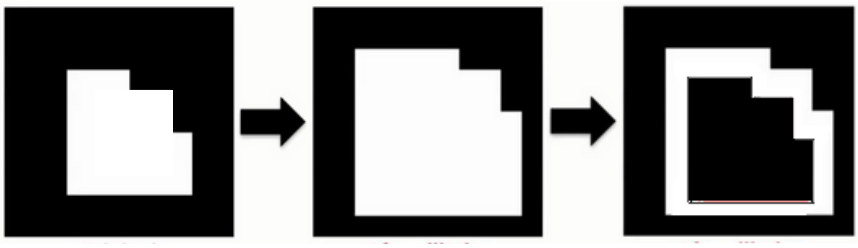
Exemple d'initialisation du pixel central avec la couleur moyenne du patch.

Méthode de synthèse d'un pixel

La fonction ***synthesize_pixel*** utilise la weighted SSD pour ***évaluer la similarité*** entre le patch synthétisé et les patches de l'image d'exemple. Le candidat avec la ***plus faible différence*** est sélectionné, et le pixel synthétique est remplacé par la couleur du candidat.

Méthode de récupération des pixels non remplis

La fonction ***get_unfilled_neighbors*** identifie les pixels non remplis dans une image synthétisée qui ont ***au moins un voisin généré***. Premièrement, elle dilate le masque des pixels déjà remplis pour ***étendre la zone à générer***.
Elle identifie ensuite les pixels non remplis ayant au moins un voisin généré en soustrayant le masque d'origine au masque dilaté. Ces pixels sont triés en fonction du nombre de voisins générés, permettant de ***prioriser les pixels qui ont davantage de voisins générés.***



Exemple de dilatation puis réduction du masque

Méthode de synthèse de texture

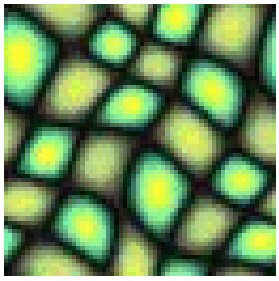
La fonction ***synthesize_texture*** coordonne l'ensemble du processus de synthèse de texture en ***itérant sur les pixels à synthétiser***. L'image synthétique est mise à jour progressivement, et le processus se poursuit jusqu'à ce que tous les pixels soient remplis.

Éxécution principale

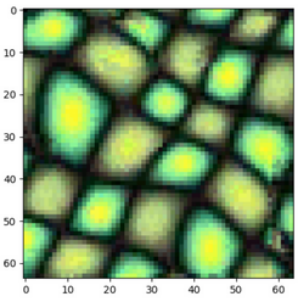
La partie principale du script instancie la classe TextureGenerator avec des ***paramètres spécifiques***, génère la texture synthétique, convertit les valeurs des pixels, et sauvegarde la texture synthétique en tant qu'image TIF.

EXEMPLES DE SYNTHÈSES

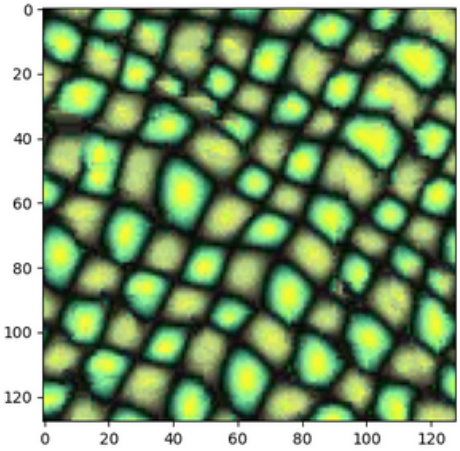
POUR TAILLE = 64X64 PUIS 128X128 ET EPSILON = 15/6



Texture d'origine 64x64



Texture synthétisée 64x64



Texture synthétisée 128x128

- Plus l'***epsilon*** (variance du masque gaussien) est bas, plus il y a de ***bruit et de variation***.
- Plus la ***taille du patch*** est élevée, plus l'on capte les ***formes/détails***.

CLASSIFIEUR DE TEXTURE

Nom des fichiers correspondant au code : KMeansClassifier.py, texture_classification.py, texture_classification_utilities.py.

Le code implémente un système de *classification de textures* en utilisant l'algorithme *K-Means.* La classe KMeansClassifier fournit une implémentation de l'algorithme, tandis que le script principal offre des démonstrations et des utilitaires pour ***estimer le nombre optimal de clusters***, ***pré-traiter les données*** de textures, et effectuer des ***classifications sur des ensembles variés*** de données. L'utilisation d'***histogrammes de gradients*** contribue au ***pré-traitement des données*** pour une meilleure représentation des textures.

COMPOSANTS DU CODE

Classe KMeansClassifier

La première partie du code concerne la classe *KMeansClassifier qui permet l'implémentation de l'algorithme de classification K-Means*. Elle ***initialise les attributs*** nécessaires tels que le nombre de clusters, le nombre maximal d'itérations, et le seuil de convergence.

- La méthode *init_fit* initialise les structures de données nécessaires et sélectionne aléatoirement des ***centroïdes*** à partir des données.
- La fonction *check_convergence_is_finished* permet de vérifier la convergence de l'algorithme en calculant la ***distance euclidienne entre les positions actuelles et précédentes des centroïdes***.
- Les méthodes *assignment_step* et *update_step* effectuent respectivement l'***assignation des points aux clusters*** et la ***mise à jour des positions des centroïdes*** en fonction des points associés à chacun d'eux.
- La méthode *fit* itére sur les étapes d'assignation et de mise à jour ***jusqu'à ce que la convergence soit atteinte***.
- La fonction *predict* associe de ***nouveaux points aux clusters*** en utilisant les centroïdes appris.

Démonstration et utilitaires

La deuxième partie du code concerne la ***démonstration et l'application du modèle K-Means sur des données de textures***, ainsi que des ***utilitaires*** pour la gestion et la visualisation des données.

- La fonction *estimate_number_of_components* utilise la silhouette score pour estimer le nombre optimal de clusters. C'est une métrique d'évaluation de la ***qualité d'un regroupement (clustering)*** des données. Il mesure à quel point chaque point d'un cluster est ***similaire*** aux autres points de ce même cluster par rapport aux points des autres clusters.

- La fonction *texture_preprocessing* effectue le ***pré-traitement des données*** en calculant les histogrammes de gradients et en standardisant les données. Dans le contexte du projet, l'*histogramme des gradients* est utilisé comme une représentation compacte des textures, contribuant à la réduction de la dimensionnalité des données tout en préservant des informations discriminantes.

- Les fonctions *demo_** sont plusieurs démonstrations montrant l'utilisation du modèle K-Means sur des **données 2D**, des ***données de textures***, des ***patches synthétiques***, et des ***patches d'images floutées***.

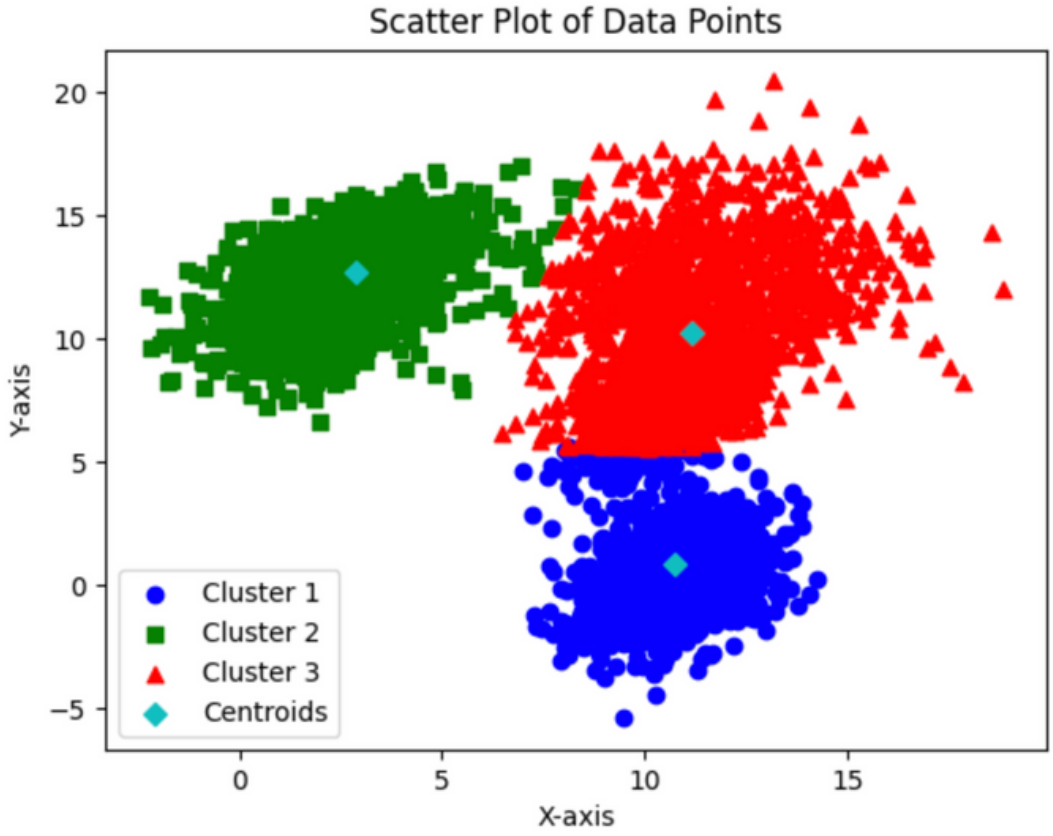
- Les fonctions *utilitaires* permettent d'extraire des patches d'images, calculer des histogrammes de gradients, et afficher des données.

EXEMPLES DE CLASSIFICATIONS

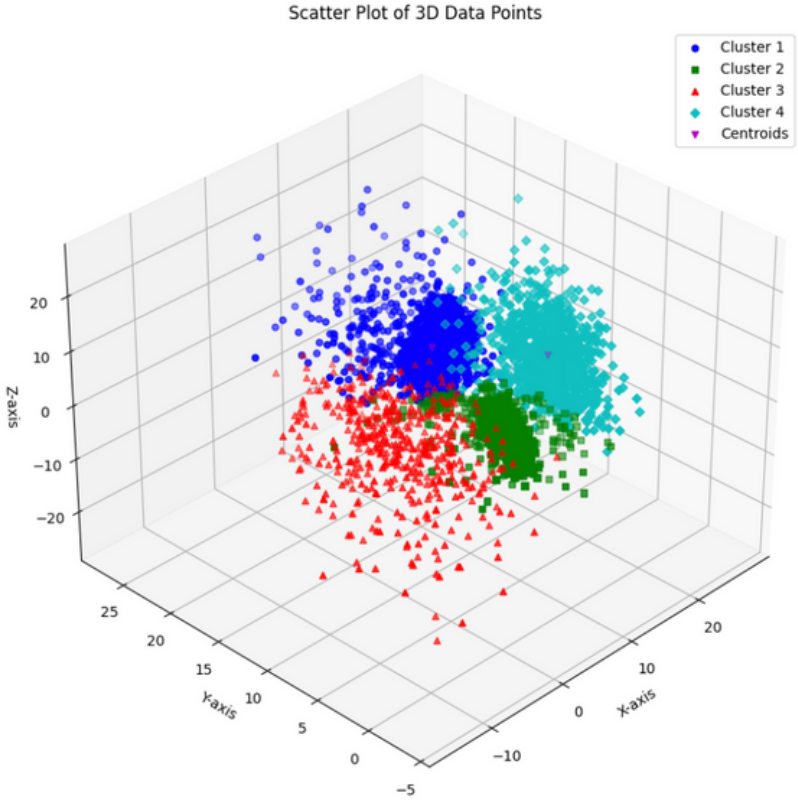
POUR LA CLASSIFICATION DE POINTS

Itérations : nombre d'itérations nécessaires pour entraîner le K-means.
Temps : temps mesurés entre le lancement du script et l'affichage d'un résultat.

	Itérations	Temps
demo_2d_kmeans	11	3.7s



	Itérations	Temps
demo_3d_kmeans	10	3.974s



Note : +40 secondes si l'on calcule le nombre de composantes optimales avant.

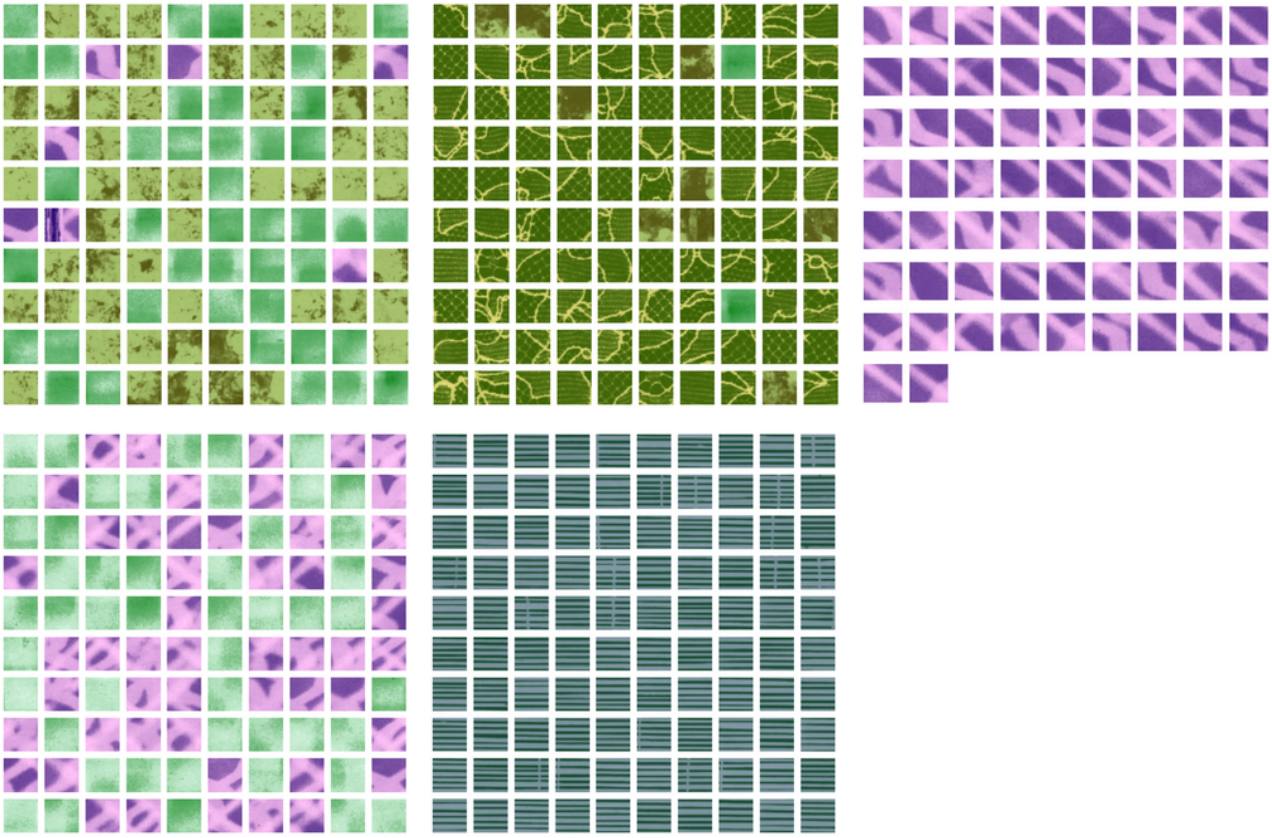
POUR LA CLASSIFICATION DE TEXTURES FOURNIES

Nombre d'images composant le dataset entier

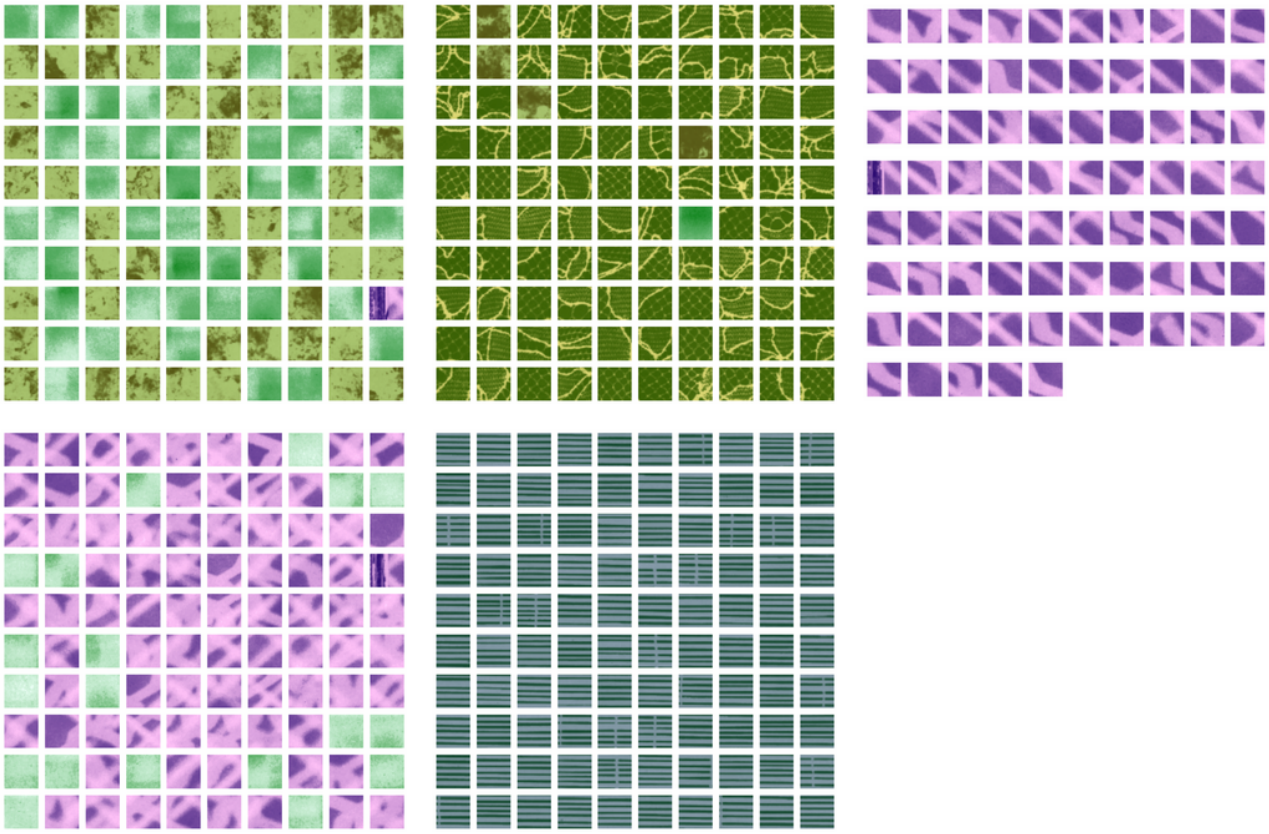
	Pour 5 images		*Pour 100 images*	
	Itérations	**Temps**	**Itérations**	**Temps**
texture_classification_gray	11	3.156s	96	2m53.056s
texture_classification_color	6	4.025s	96	4m7.182s
texture_classification_blurred	11	4.260s	107	3m55.120s

Note : Blurred = color flouté.

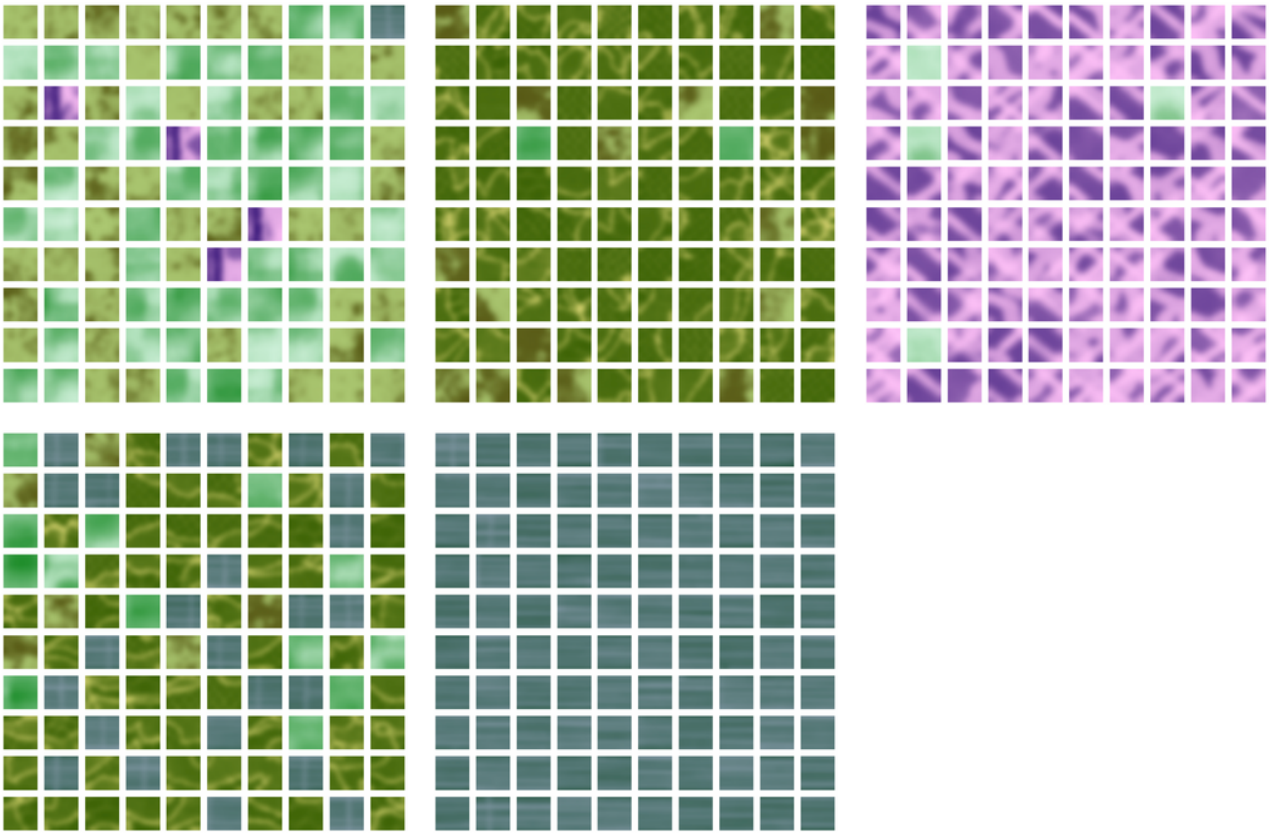
Pour 5 images - texture_classification_gray



Pour 5 images - texture_classification_color



Pour 5 images - texture_classification_blurred



Remarques :

- La classification de textures en couleur prend ***plus de temps parce qu'on a 3 fois plus de données*** (3 channels de couleurs = 3 histogrammes de gradients).

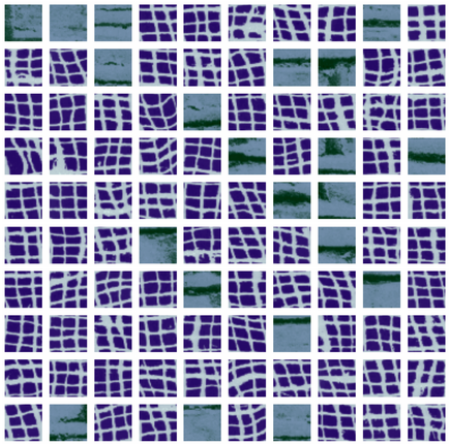
- La classification de textures floues prend ***autant de temps que la classification couleur puisqu'il y a autant de données*** (même si c'est flouté on a toujours 3 channels).

POUR LA CLASSIFICATION D'UNE TEXTURE SYNTHÉTISÉE

Nombre d'images composant le dataset d'entraînement

	Pour 5 images		Pour 50 images	
	Itérations	**Temps**	**Itérations**	**Temps**
synthesized_patch_classification	8	4.369s	128	3m57.04s

Pour 50 images - texture_patch_classification



Remarques :

- Le nombre d'images est de 50 pour le second test car ici le nombre d'images correspond à la ***taille du dataset d'entraînement***, alors qu'auparavant le dataset était scindé en 2 pour avoir un set d'entraînement et un set de test.

- On constate que le patch synthétisé est classé dans le ***même cluster que beaucoup de patches issus de la texture d'origine***. Ceci indique que notre algorithme de génération de texture est relativement bon.

Remarques générales :

- La ***couleur influence les performances*** en temps de l'algorithme.
- Lorsque les textures sont floues, il a besoin de ***plus d'itérations pour séparer les clusters*** mais les itérations ont l'air de coûter moins cher puisque le temps est similaire à la classification non floutée.
- Il faut ***plus d'itérations et de temps pour trier plus d'images***.

Résultats :

- Un set de patches (une image) correspond à un ***cluster pris au hasard parmis les n clusters détectés*** dans lequel on a pris jusqu'à 100 patches au hasard, cela permet de donner une idée globale du contenu du cluster (et chaque image est divisée en 100 donc chaque cluster devrait idéalement contenir exactement 100 patches, l'objectif étant d'avoir un cluster par image).
- Les patches ne sont pas parfaitement classés mais les patches d'un ***même cluster se ressemblent*** en général.
- La prise en compte de la ***couleur améliore un peu la précision***.
- Plus il y a d'images dans le dataset et ***moins l'on est précis*** (puisqu'il va être plus difficile de séparer les données, surtout si il y a des images similaires dans le dataset et c'est le cas ici).
- Pour améliorer la précision on pourrait trouver une ***autre manière de représenter les patches*** qui permettrait de mieux les différencier.