

Segmentation d'image par marches aléatoires

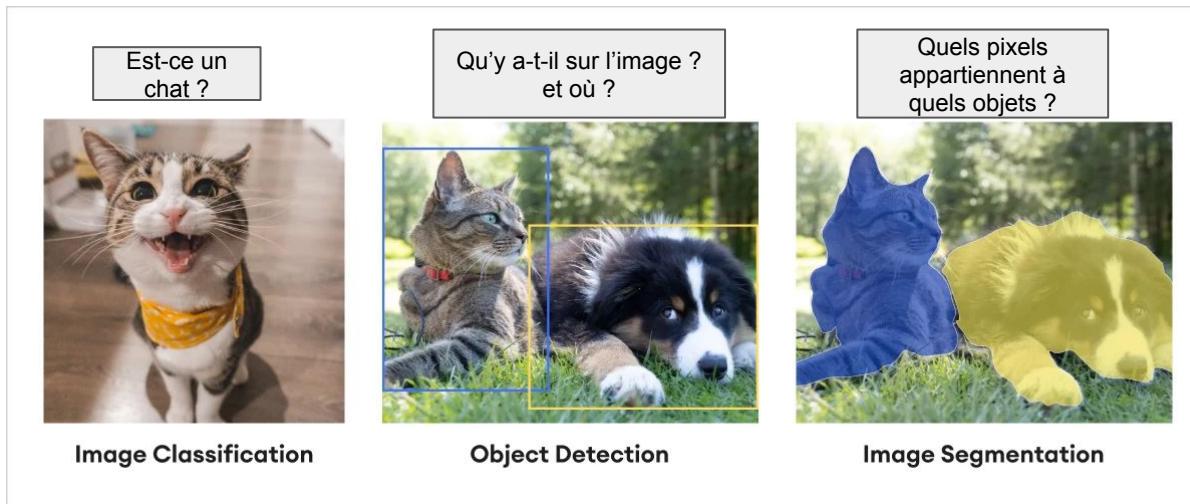
TIPE 2024-25 - Transition, transformation,
conversion

COURCELLE Dorian
SCEI : _

Introduction

❖ Qu'est ce que la segmentation d'image ?

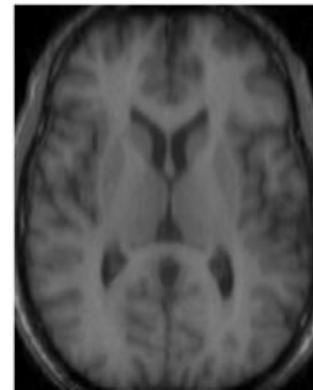
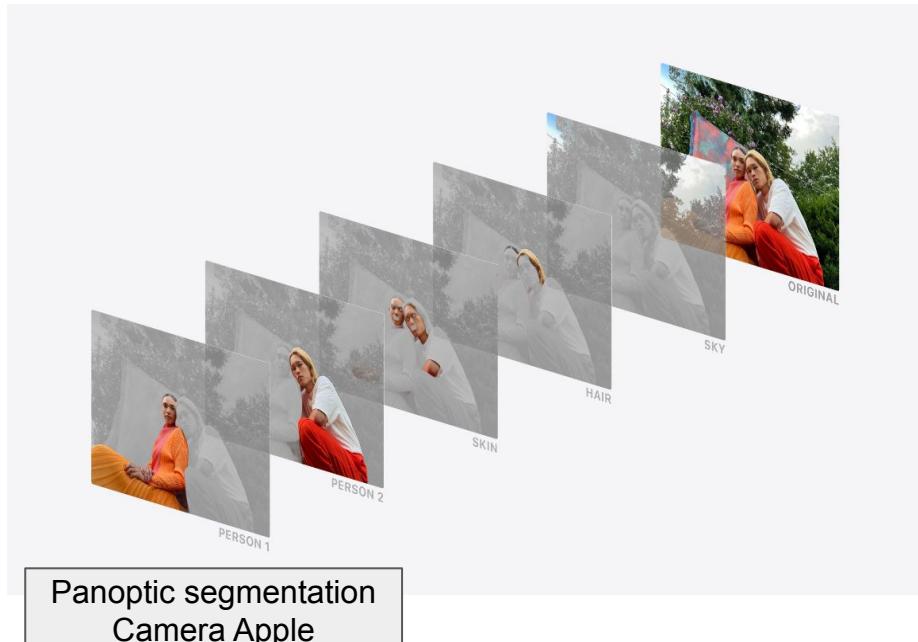
La segmentation d'image est une technique de vision par ordinateur qui divise une image numérique en groupes de pixels distincts (segments d'image) afin de faciliter la détection d'objets et les tâches connexes (IBM)



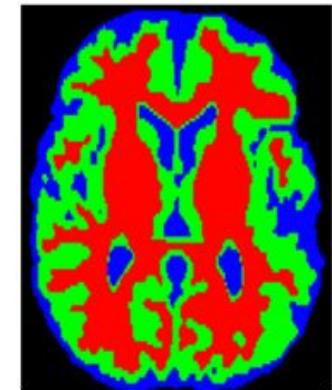
Différents problèmes de vision par ordinateur

Introduction

- ❖ À quoi ça sert ?



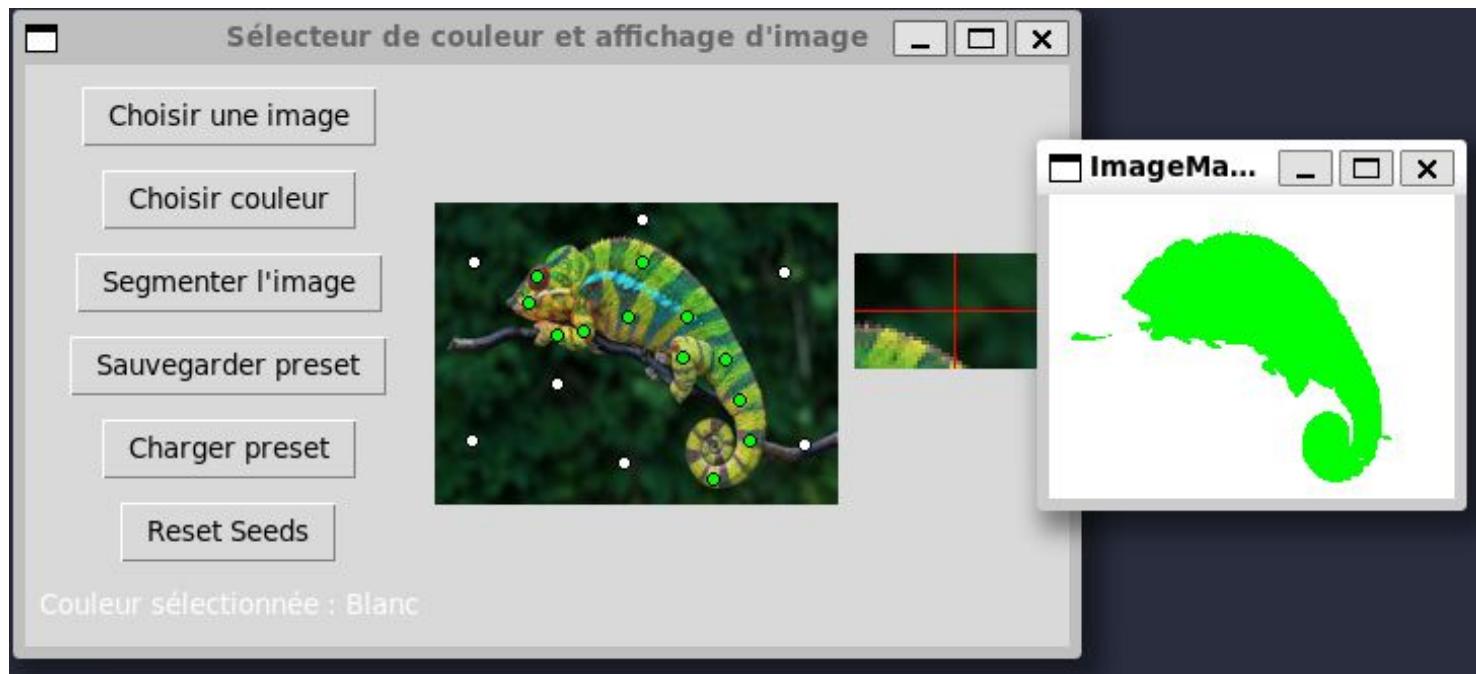
(a) coupe axial



(b) segmentation des tissus

Une segmentation d'une coupe axiale de cerveau. Le rouge, le vert et le bleu correspondent respectivement à la substance blanche, à la substance grise et au liquide céphalo-rachidien. (Source de l'image : *A review of medical image segmentation: methods and available software*)

L'algorithme de segmentation par marches aléatoires : principe



Interface graphique tkinter pour mon algorithme de segmentation

L'algorithme de segmentation par marches aléatoires est dit semi-automatique

Modélisation d'une image par un graphe

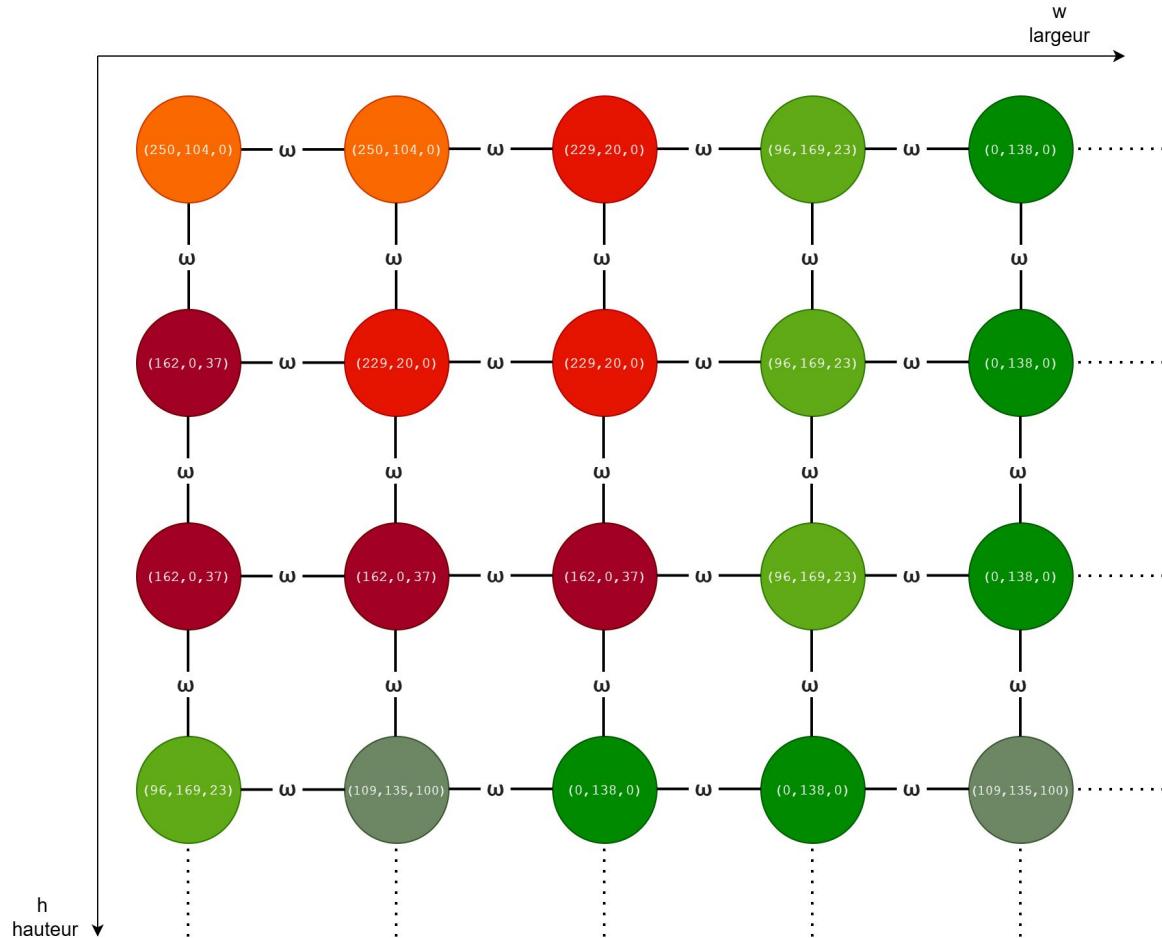


Illustration de l'algorithme naïf

❖ Choix des seeds :

L'utilisateur marque deux sommets (pixels) comme étant des seeds :

- une seed le label 1
- une seed de label 2

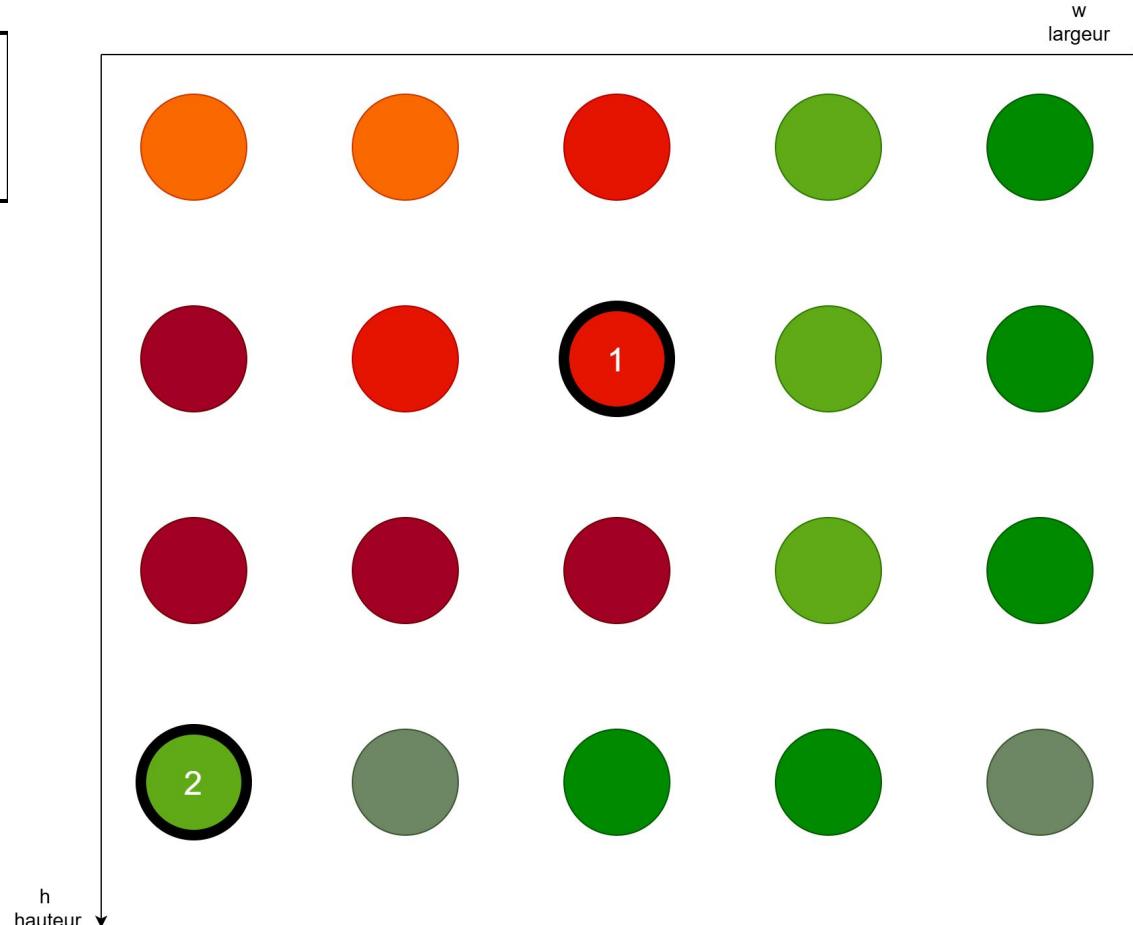


Illustration de l'algorithme naïf

- ❖ Tirage aléatoire de pixel :

L'algorithme tire aléatoirement un pixel non marqué.

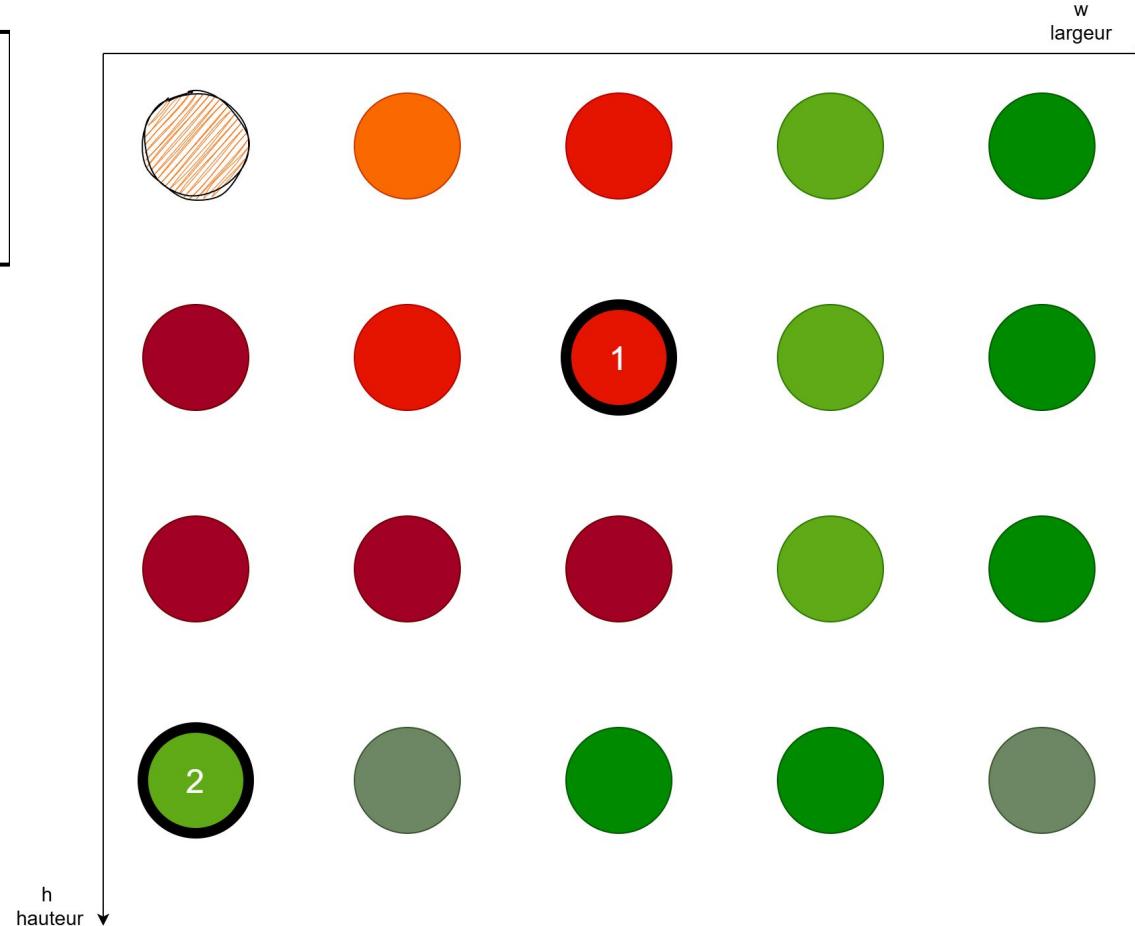


Illustration de l'algorithme naïf

- ❖ Marche aléatoire :

L'algorithme effectue une **marche aléatoire** depuis le sommet sélectionné précédemment jusqu'à **convergence** vers un sommet seed (sommet absorbant).

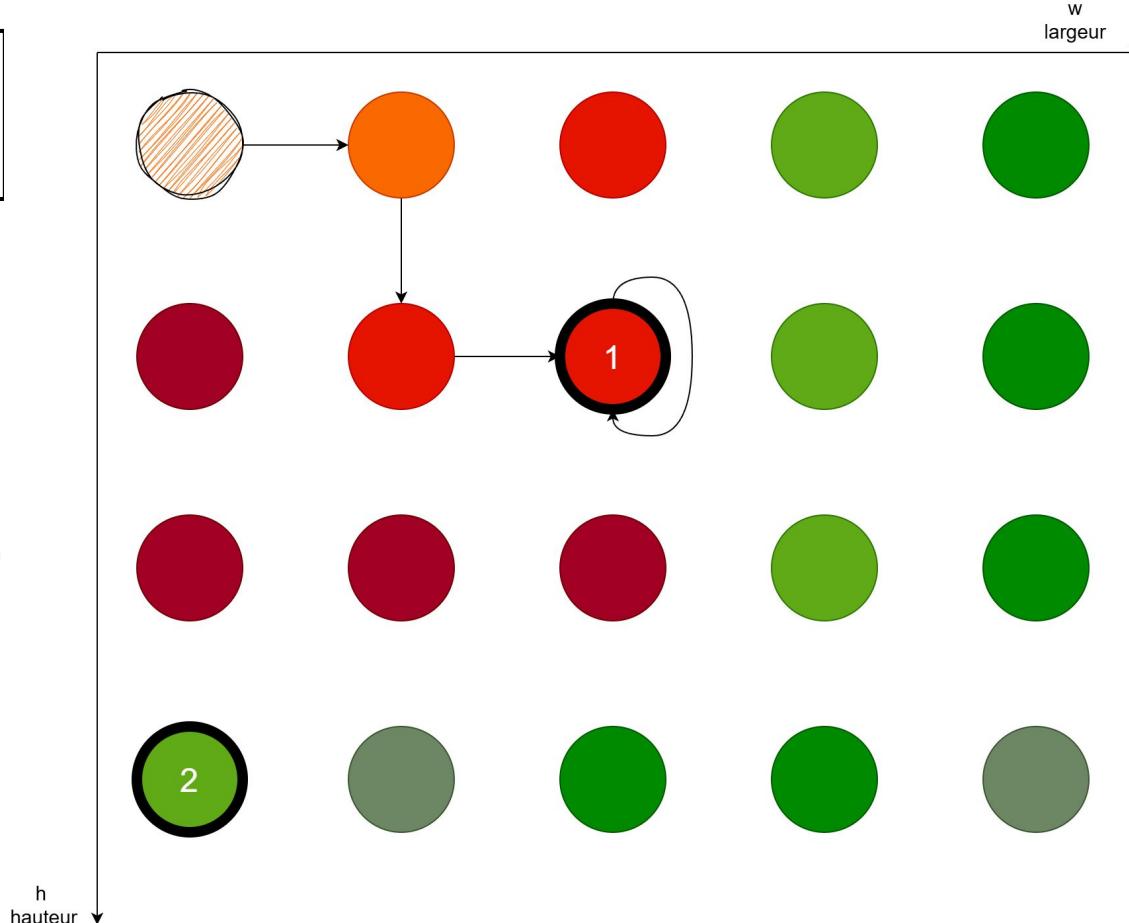


Illustration de l'algorithme naïf

- ❖ Marquage d'une nouvelle seed :

Le sommet de départ est **marqué du même label** que celui du sommet vers lequel la marche aléatoire a convergé.

Celui-ci devient alors une seed.

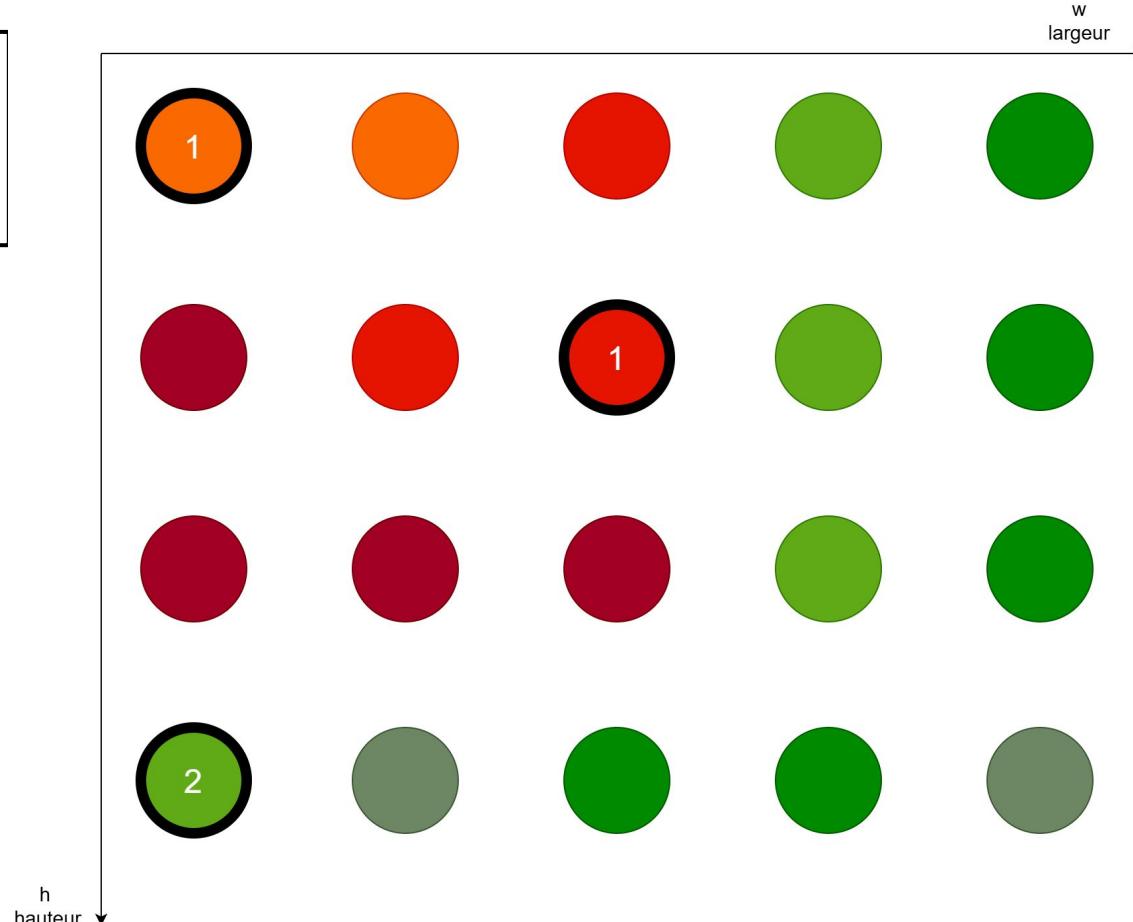


Illustration de l'algorithme naïf

- ❖ Tirage aléatoire de pixel :

L'algorithme tire aléatoirement un pixel non marqué.

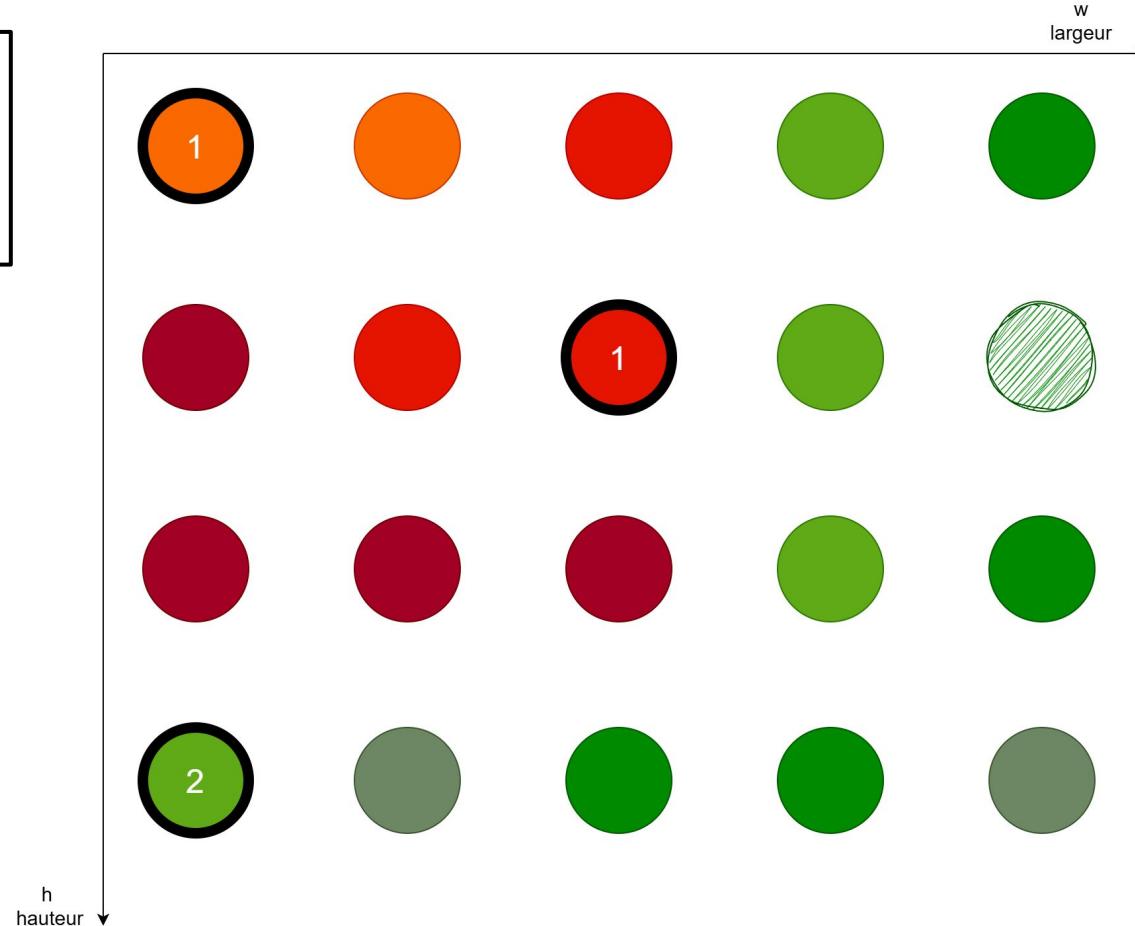


Illustration de l'algorithme naïf

- ❖ Marche aléatoire :

L'algorithme effectue une **marche aléatoire** depuis le sommet sélectionné précédemment jusqu'à **convergence** vers un sommet seed (sommet absorbant).

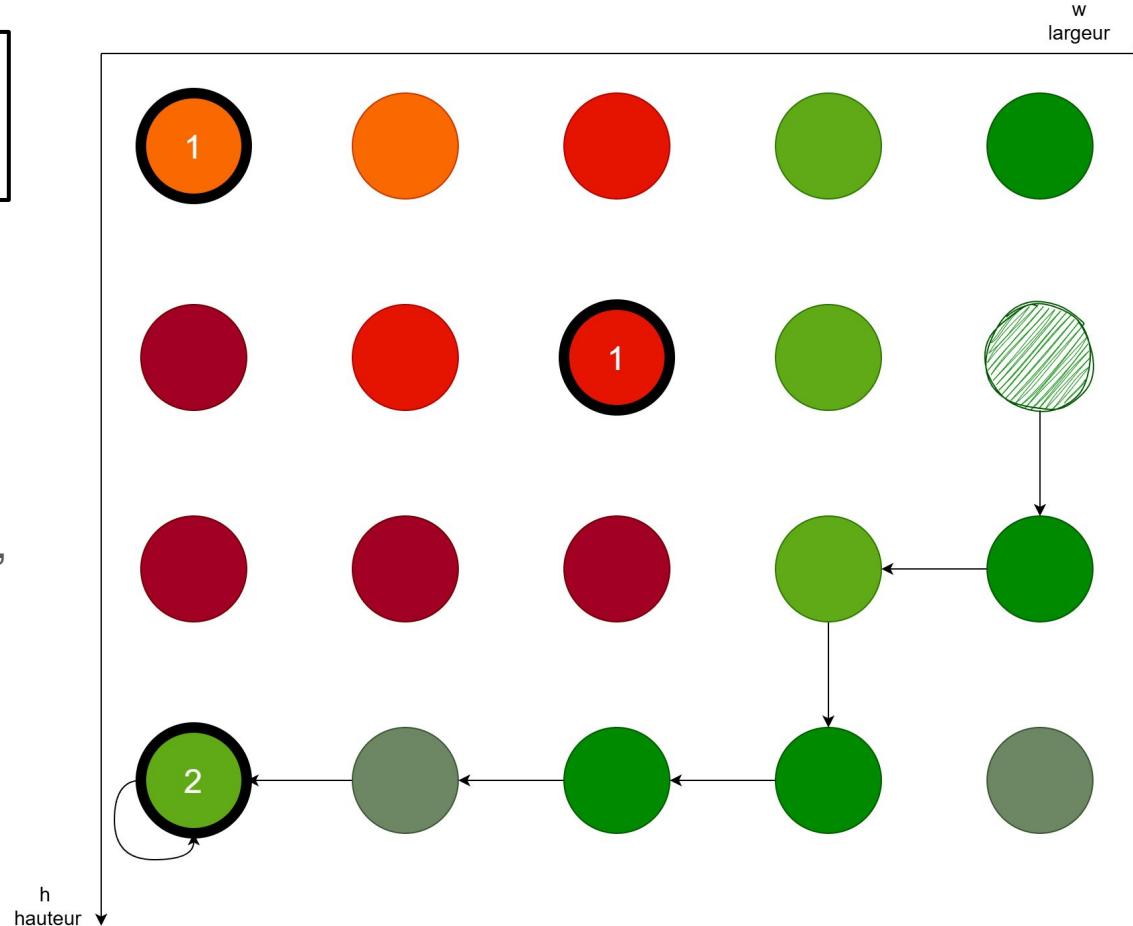


Illustration de l'algorithme naïf

- ❖ Marquage d'une nouvelle seed :

Le sommet de départ est **marqué du même label** que celui du sommet vers lequel la marche aléatoire a convergé.

Celui-ci devient alors une seed.

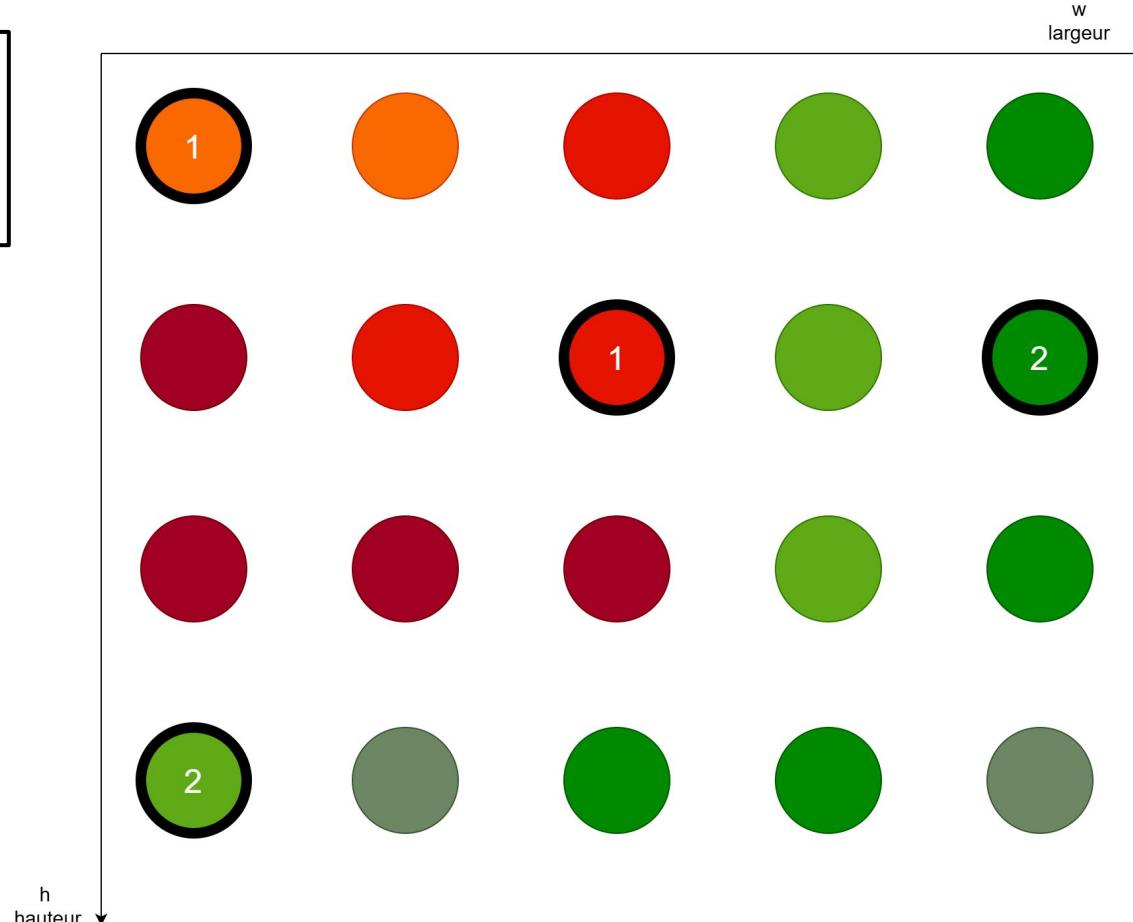
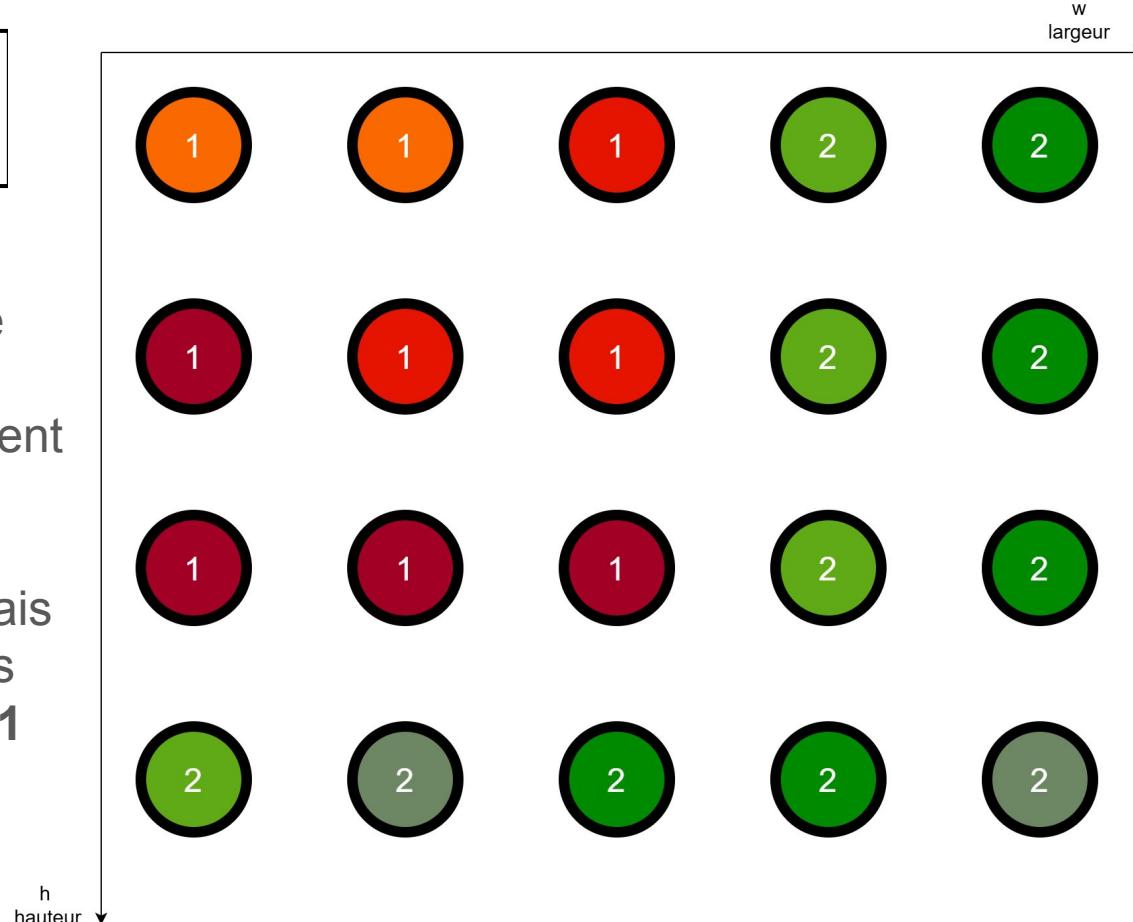


Illustration de l'algorithme naïf

- ❖ Segmentation finale :

On continue le même procédé jusqu'à que tous les sommets soient marqués.

On distingue désormais deux ensembles dans l'image : l'**ensemble 1** et le l'**ensemble 2**.



Problématique

L'algorithme de segmentation par marches aléatoires étant coûteux en calcul, son optimisation passe par l'analogie électrique pour accélérer le processus. On pourrait également passer d'un modèle semi-automatique à un modèle entièrement automatique grâce à l'apprentissage supervisé, mais quel en serait l'impact sur les performances ?

Table des matières

- 1. Résultats de l'algorithme naïf et améliorations**
 - a. **Marche non convergente et amélioration de la précision**
 - b. **Différents calculs de poids pour s'adapter aux textures**
2. Analogie circuit électrique : une tentative d'optimisation
 - a. Présentation de l'algorithme
 - b. Résultats
3. Automatisation de la sélection des seeds et résultats
 - a. Construction du réseau de neurones
 - b. Dataset et entraînement
 - c. Résultats
4. Conclusion

Marche non convergente et amélioration de la précision

```
def marche_aleatoire(self, origin):
    # On commence la marche depuis le pixel 'origin'
    cur_pixel = origin

    # On récupère les voisins de ce pixel dans le graphe pondéré (graph_image)
    neighbors = self.graphe_image.get_neighbors(cur_pixel)

    i = 0 # Compteur d'étapes
    while i < self.max_steps:
        i += 1

        # Choix probabiliste d'un voisin :
        # Chaque voisin est choisi avec une probabilité proportionnelle au poids de l'arête
        cur_pixel = random.choices(
            population=[neighbor[0] for neighbor in neighbors], # Liste des voisins
            weights=[neighbor[1] for neighbor in neighbors] # poids associés
        )[0]

        # Si on atteint un pixel déjà segmenté (ayant une étiquette non nulle)
        if self.segmented_image[cur_pixel[0]][cur_pixel[1]] != 0:
            self.marche_convergente += 1 # Compteur de marches ayant atteint une zone déjà classée

            # On récupère l'étiquette/couleur de ce pixel déjà étiqueté
            label = self.segmented_image[cur_pixel[0]][cur_pixel[1]]

            # On l'applique au pixel d'origine de la marche
            self.segmented_image[origin[0]][origin[1]] = label
            break # Marche terminée

        # Sinon, on continue la marche depuis ce nouveau pixel
        neighbors = self.graphe_image.get_neighbors(cur_pixel)

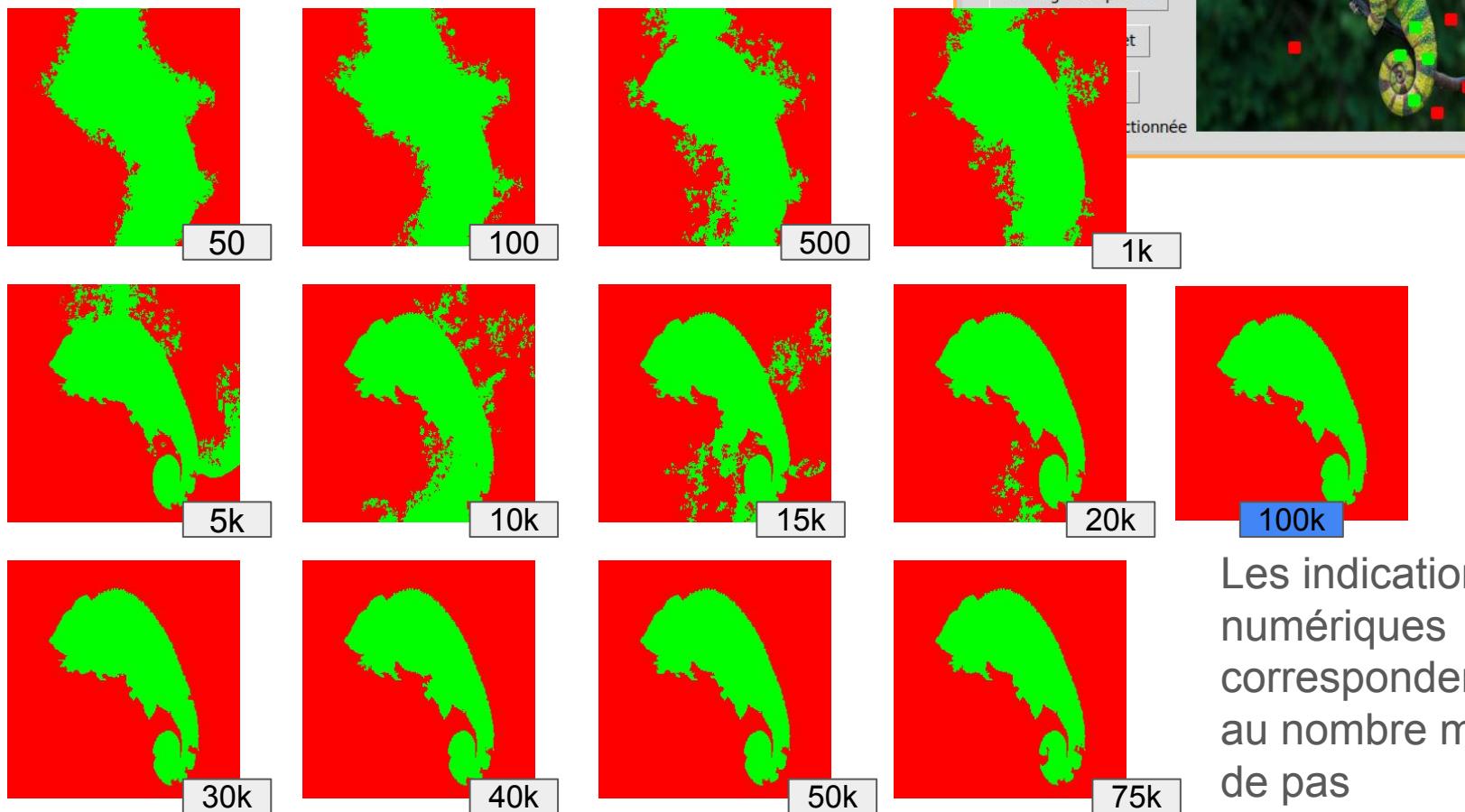
    # Si on atteint le nombre max d'étapes sans croiser une zone déjà segmentée :
    if i == self.max_steps:
        # On attribue au pixel 'origin' l'étiquette de la seed (graine) la plus proche géométriquement
        min_dist = float("inf")
        min_seed = None
        for seed in self.seeds:
            dist = sqrt((origin[0] - seed[0])**2 + (origin[1] - seed[1])**2)
            if dist < min_dist:
                min_dist = dist
                min_seed = seed

        # On applique l'étiquette de cette seed au pixel
        self.segmented_image[origin[0]][origin[1]] = self.seeds[min_seed]
```

Influence du nombre maximal de marches sur la qualité de la segmentation

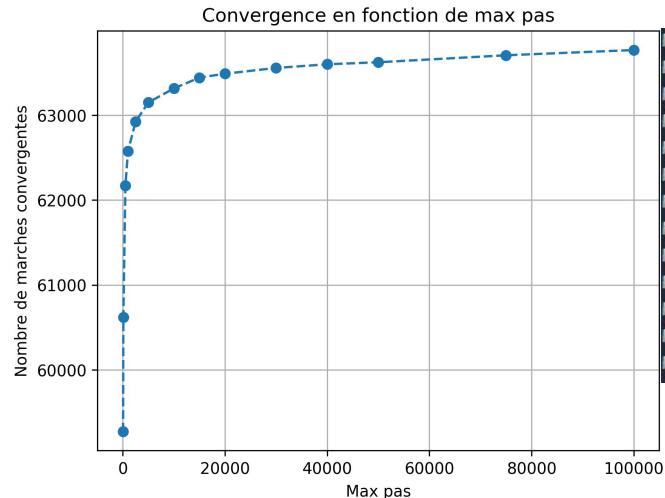
En utilisant les mêmes seeds :

Preset 'ameleon_256' chargé (784 points).



Les indications numériques correspondent au nombre max de pas

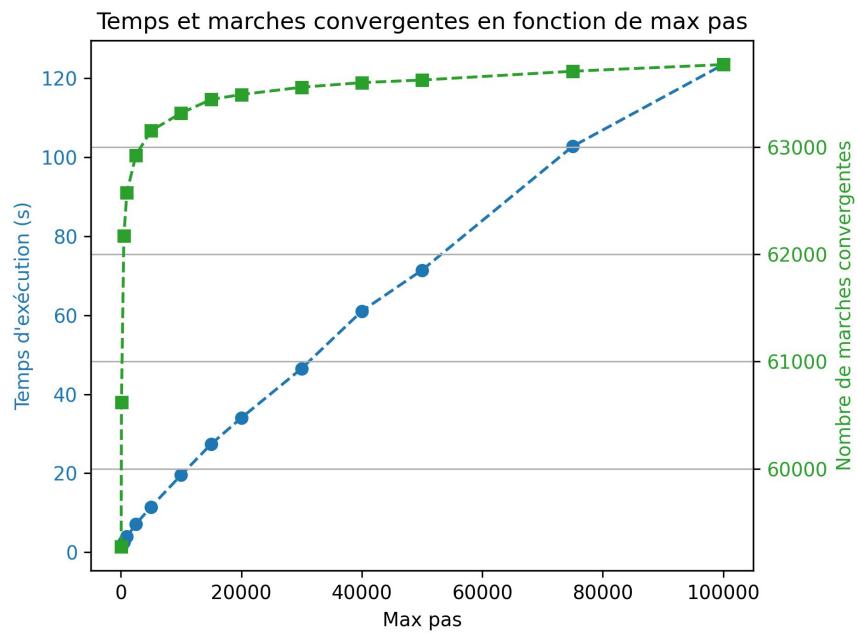
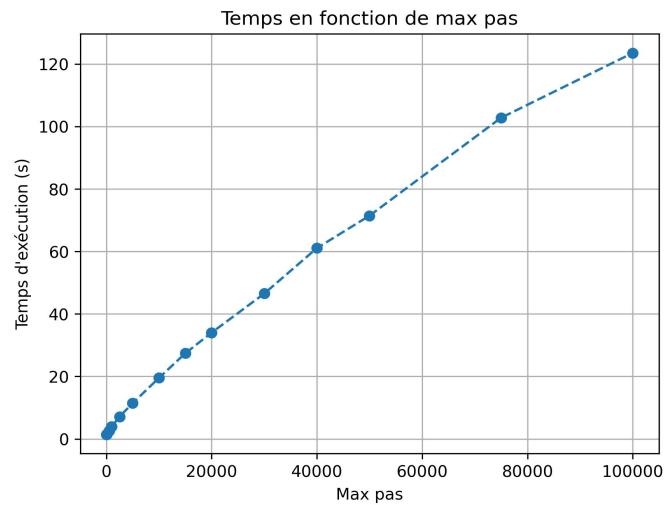
Influence du nombre maximal de marches sur la qualité de la segmentation



Max pas : 50 Nb convergentes : 59,275
 Max pas : 100 Nb convergentes : 60,620
 Max pas : 500 Nb convergentes : 62,170
 Max pas : 1,000 Nb convergentes : 62,576
 Max pas : 5,000 Nb convergentes : 63,151
 Max pas : 10,000 Nb convergentes : 63,316
 Max pas : 15,000 Nb convergentes : 63,443
 Max pas : 20,000 Nb convergentes : 63,489
 Max pas : 30,000 Nb convergentes : 63,557
 Max pas : 40,000 Nb convergentes : 63,600
 Max pas : 50,000 Nb convergentes : 63,624
 Max pas : 75,000 Nb convergentes : 63,706
 Max pas : 100,000 Nb convergentes : 63,768

max marches convergentes :

$$256 \times 256 - 784 = \\ 64\,752$$



Différents calculs de poids pour s'adapter aux textures

Méthode 1 : distance euclidienne des valeurs RGB :



METHODE 1

$$\text{distance} = \sqrt{(96 - 0)^2 + (169 - 80)^2 + (23 - 239)^2}$$

$$\text{poids} = \exp\left(-\frac{\text{distance}^2}{\sigma^2}\right)$$

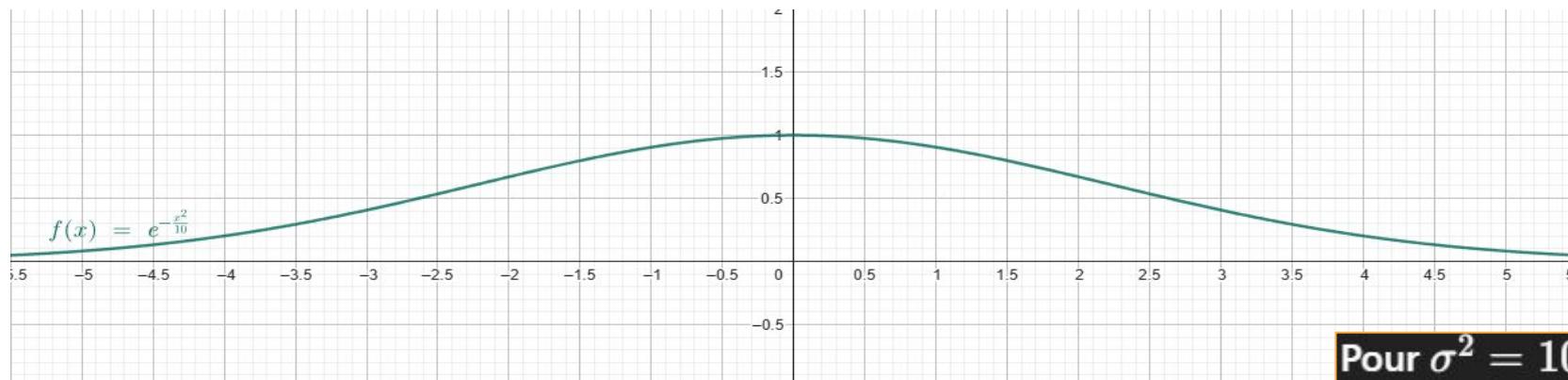
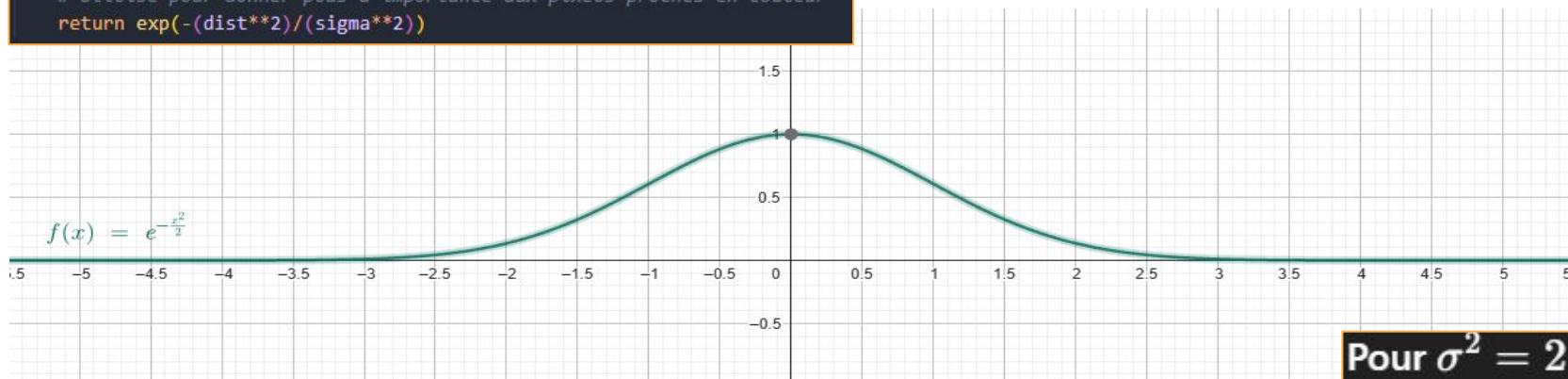
```
def distance_couleur(self, p1, p2):
    # Récupère les couleurs RGB des deux pixels p1 et p2 dans l'image
    rgb1 = self.img_array[p1[0], p1[1]].astype(np.int16)
    rgb2 = self.img_array[p2[0], p2[1]].astype(np.int16)

    # Calcule la distance euclidienne entre les deux couleurs (dans l'espace RGB)
    distance = np.linalg.norm(np.array(rgb1) - np.array(rgb2))
    return distance
```

```
def poids_gauss(self, dist, sigma):
    # Calcule un poids selon une loi gaussienne en fonction de la distance
    # Utilisé pour donner plus d'importance aux pixels proches en couleur
    return exp(-(dist**2)/(sigma**2))
```

Fonction gaussienne pour pondérer la distance

```
def poids_gauss(self, dist, sigma):
    # Calcule un poids selon une loi gaussienne en fonction de la distance
    # Utilisé pour donner plus d'importance aux pixels proches en couleur
    return exp(-(dist**2)/(sigma**2))
```



Différents calculs de poids pour s'adapter aux textures

Méthode 2 : distance euclidienne de la moyenne des valeurs RGB autour du pixel

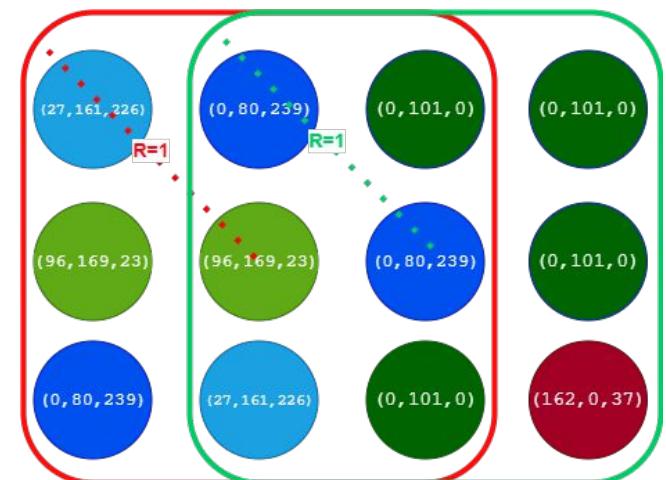
```
def moyenne_patch(self, y, x, rayon=2):
    # Définir les bornes du patch autour du pixel (y, x)
    y_min = max(0, y - rayon)
    y_max = min(self.height, y + rayon + 1)
    x_min = max(0, x - rayon)
    x_max = min(self.width, x + rayon + 1)

    # Extraction du patch autour du pixel (une sous-image)
    patch = self.img_array[y_min:y_max, x_min:x_max]

    # Retourne la moyenne RGB du patch, calculée sur les deux dimensions spatiales
    return patch.mean(axis=(0, 1)) # moyenne RGB du patch
```

```
def distance_couleur(self, p1, p2):
    desc1 = self.moyenne_patch(p1[0], p1[1], rayon=6)
    desc2 = self.moyenne_patch(p2[0], p2[1], rayon=6)
    distance = np.linalg.norm(desc1 - desc2)
    return distance
```

```
def poids_gauss(self, dist, sigma):
    # Calcule un poids selon une loi gaussienne en fonction de la distance
    # Utilisé pour donner plus d'importance aux pixels proches en couleur
    return exp(-(dist**2)/(sigma**2))
```

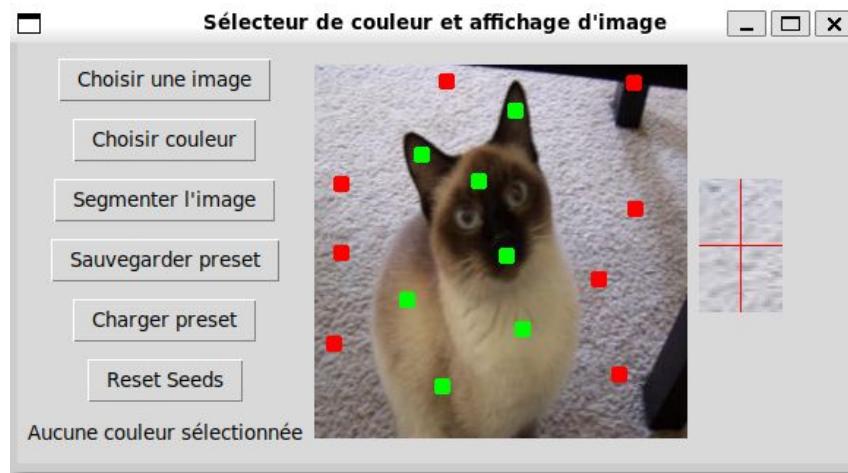


METHODE 2

$$\text{distance} = \sqrt{(\text{moyenne1} - \text{moyenne2})^2}$$

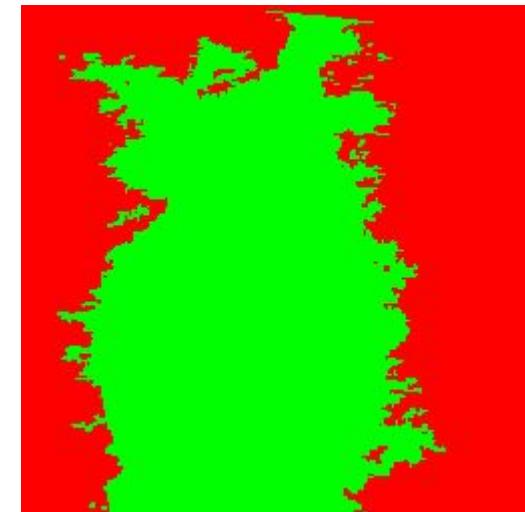
$$\text{poids} = \exp\left(-\frac{\text{distance}^2}{\sigma^2}\right)$$

Différents calculs de poids pour s'adapter aux textures

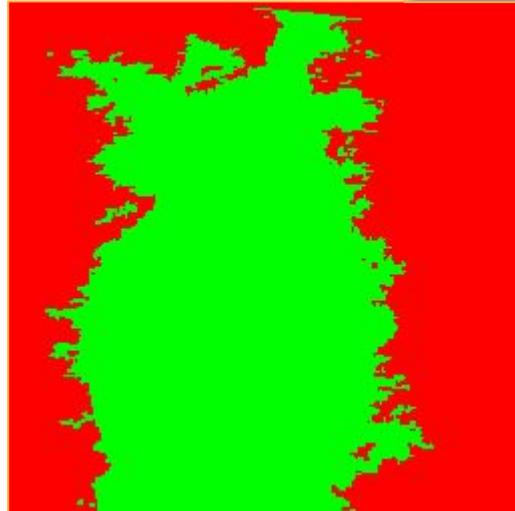
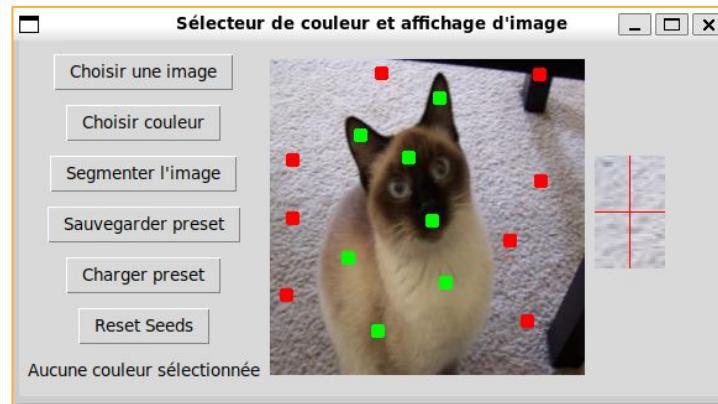


texture des poils + arrière plan similaire
à objet

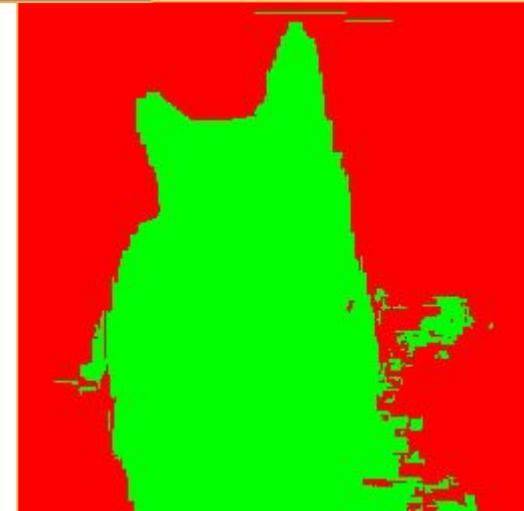
rend difficile la segmentation avec les
poids de la méthode 1



Différents calculs de poids pour s'adapter aux textures



Méthode 1 : avec sigma=6
et max_pas = 30,000
durée : 64.9283

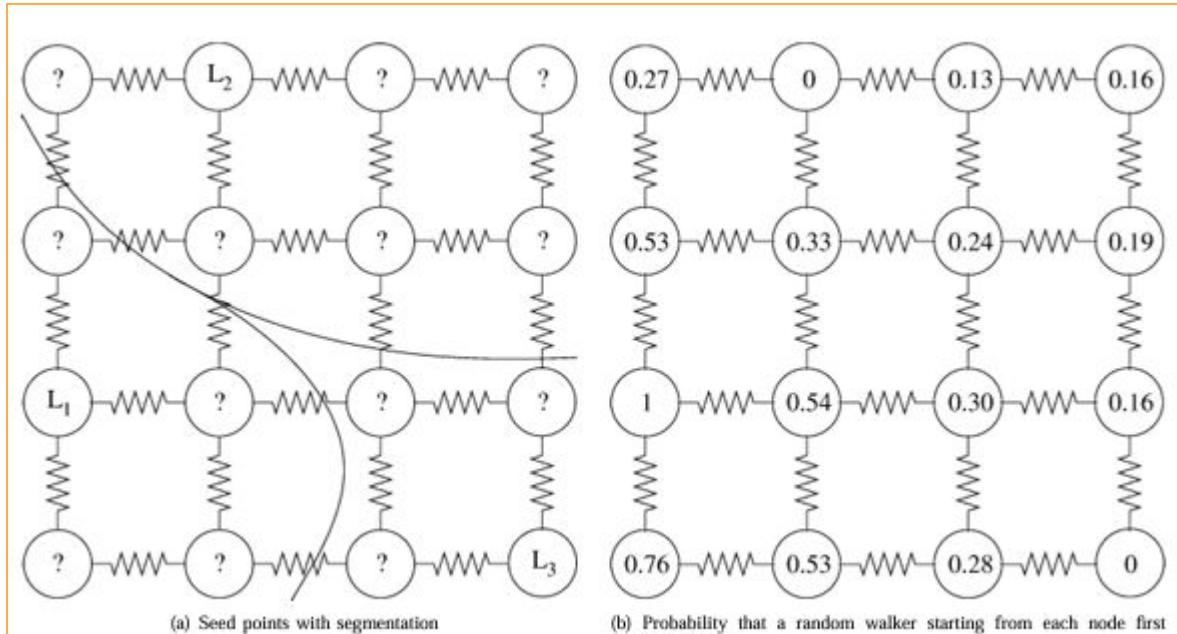


Méthode 2 : avec sigma=2
et max_pas = 80,000
durée : 40.4228

Table des matières

- 1) Résultats de l'algorithme naïf et améliorations
 - a) Marche non convergente et amélioration de la précision
 - b) Différents calculs de poids pour s'adapter aux textures
- 2) Analogie circuit électrique : une tentative d'optimisation**
 - a) Présentation de l'algorithme
 - b) Résultats
- 3) Automatisation de la sélection des seeds et résultats
 - a) Construction du réseau de neurones
 - b) Dataset et entraînement
 - c) Résultats
- 4) Conclusion

Approche électrique



extrait de [1],
Leo Grady

« Fixez alternativement le potentiel de chaque étiquette (label) à l'unité (c'est-à-dire en la reliant à une source de tension par rapport à la masse), et fixez à zéro (c'est-à-dire à la masse) les autres nœuds. Les potentiels électriques calculés représentent alors la probabilité qu'une marche aléatoire partant de chaque nœud atteigne en premier le point source actuellement fixé à 1. »

Résultats

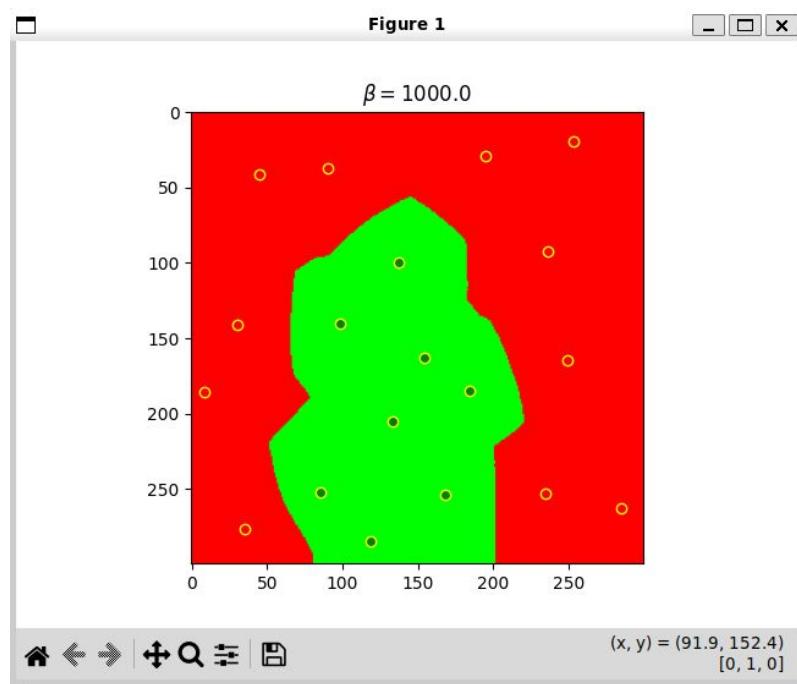
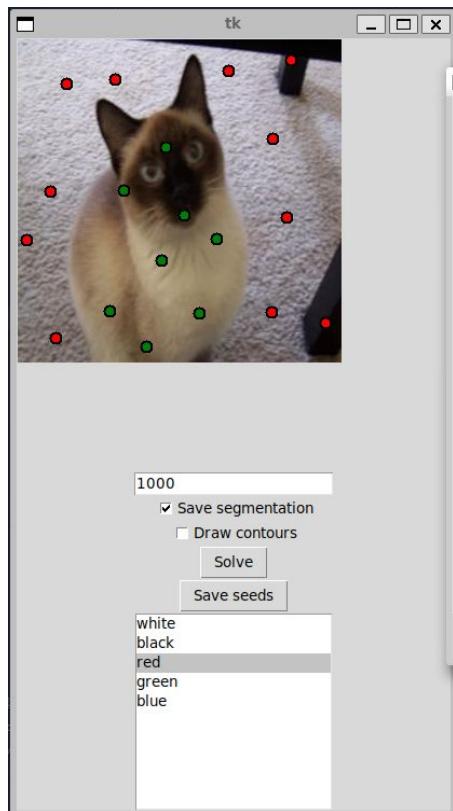
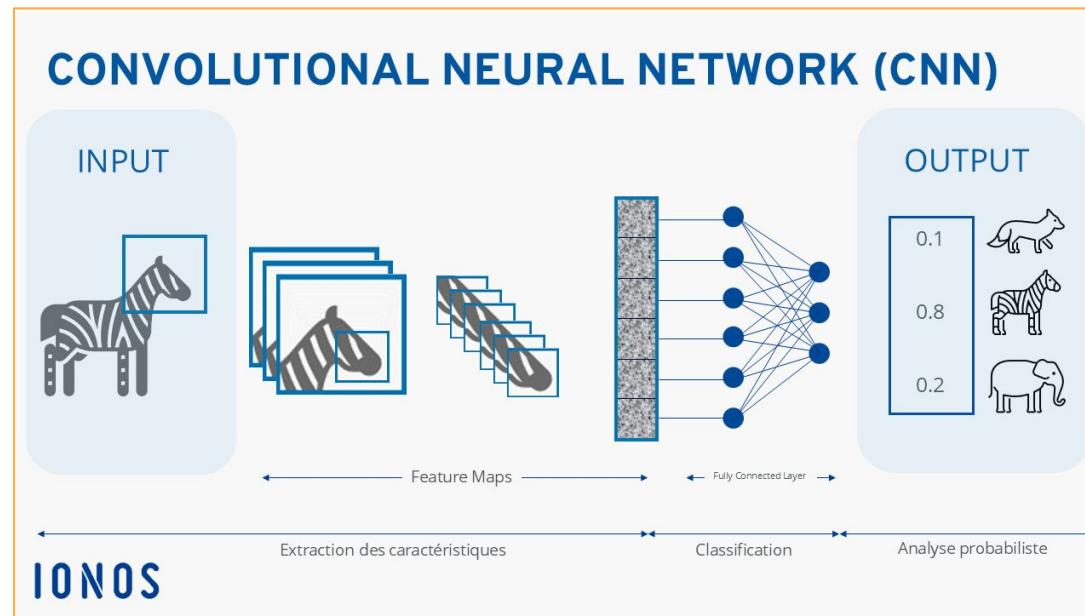


Table des matières

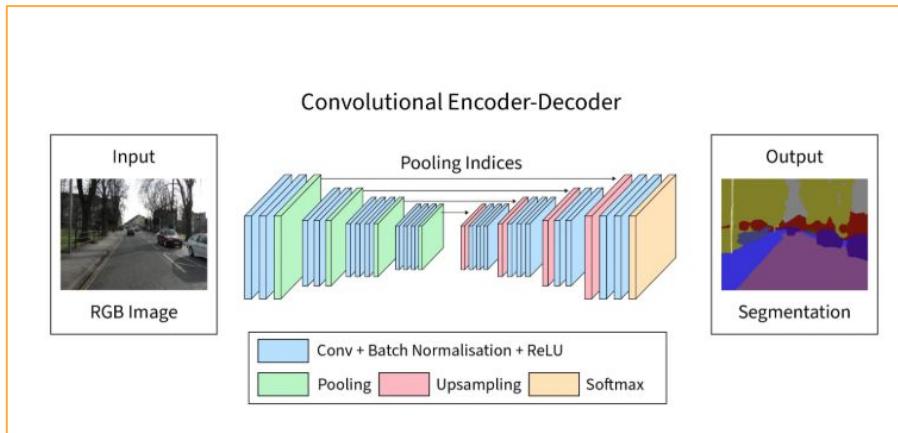
- 1) Résultats de l'algorithme naïf et améliorations
 - a) Marche non convergente et amélioration de la précision
 - b) Différents calculs de poids pour s'adapter aux textures
- 2) Analogie circuit électrique : une tentative d'optimisation
 - a) Présentation de l'algorithme
 - b) Résultats
- 3) **Automatisation de la sélection des seeds et résultats**
 - a) **Construction du réseau de neurones**
 - b) **Dataset et entraînement**
 - c) **Résultats**
- 4) Conclusion

Construction du réseau de neurones

 PyTorch



Construction du réseau de neurones



```
class MiniSeedNet(nn.Module):
    def __init__(self):
        super(MiniSeedNet, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 16, 3, padding=1), # -> [B, 16, 256, 256] couche convolutive : extrait les motifs locaux
            nn.ReLU(),
            nn.MaxPool2d(2),           # -> [B, 16, 128, 128] # divise la taille de l'image en 2 en découplant en 2*2 et gardant pixel plus grande valeur

            nn.Conv2d(16, 32, 3, padding=1), # -> [B, 32, 128, 128]
            nn.ReLU(),
            nn.MaxPool2d(2),           # -> [B, 32, 64, 64]
        )

        self.decoder = nn.Sequential(
            # le décodeur restitue une carte des seeds à la même taille que l'image
            nn.ConvTranspose2d(32, 16, 2, stride=2), # -> [B, 16, 128, 128]
            nn.ReLU(),
            nn.ConvTranspose2d(16, 16, 2, stride=2), # -> [B, 16, 256, 256]
            nn.ReLU(),
            nn.Conv2d(16, 2, 1)                  # -> [B, 2, 256, 256] B = batchsize 2 = canaux de sortie (fond(0)/objet(1)) (256,256) h*w
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x # Logits pour chaque classe
```

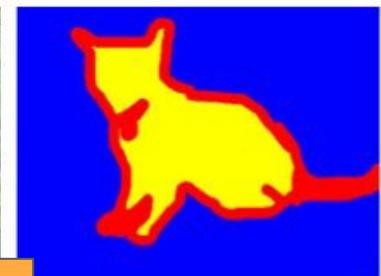
Dataset utilisé

Dataset utilisé : Oxford-IIIT Pet Dataset

Avec sélection uniquement des images de chats : **636** pour l'entraînement et **159** pour l'évaluation du modèle. (répartition 80|20)

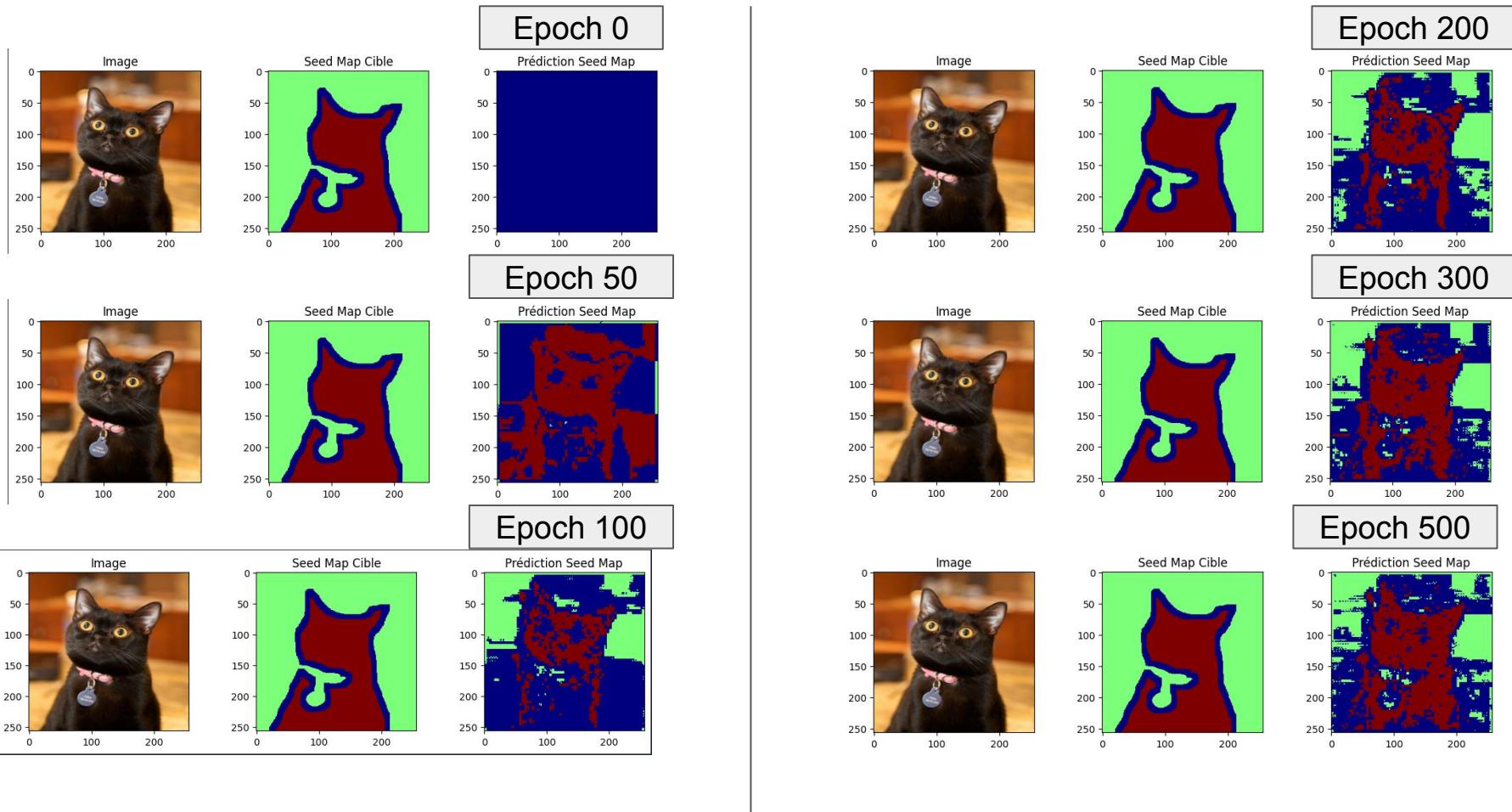
Breed	Count
Abyssinian	198
Bengal	200
Birman	200
Bombay	200
British Shorthair	184
Egyptian Mau	200
Main Coon	190
Persian	200
Ragdoll	200
Russian Blue	200
Siamese	199
Sphynx	200
Total	2371

2.Cat Breeds



Exemple
d'annotation du
dataset

Progression de l'entraînement



```
pred_seed_map[fg_probs > 0.85] = 1  
pred_seed_map[bg_probs > 0.85] = 0
```

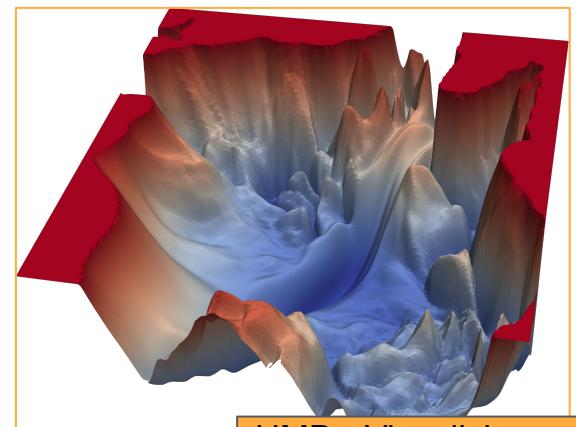
Entrainement : loss function et optimizer

la loss function permet de déterminer le coût, ce dernier nous indiquant la proximité de la prédiction en sortie par rapport au résultat attendu.

```
weights = torch.tensor([1.0, 5.0]).to(device)
criterion = nn.CrossEntropyLoss(ignore_index=-1, weight=weights)
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

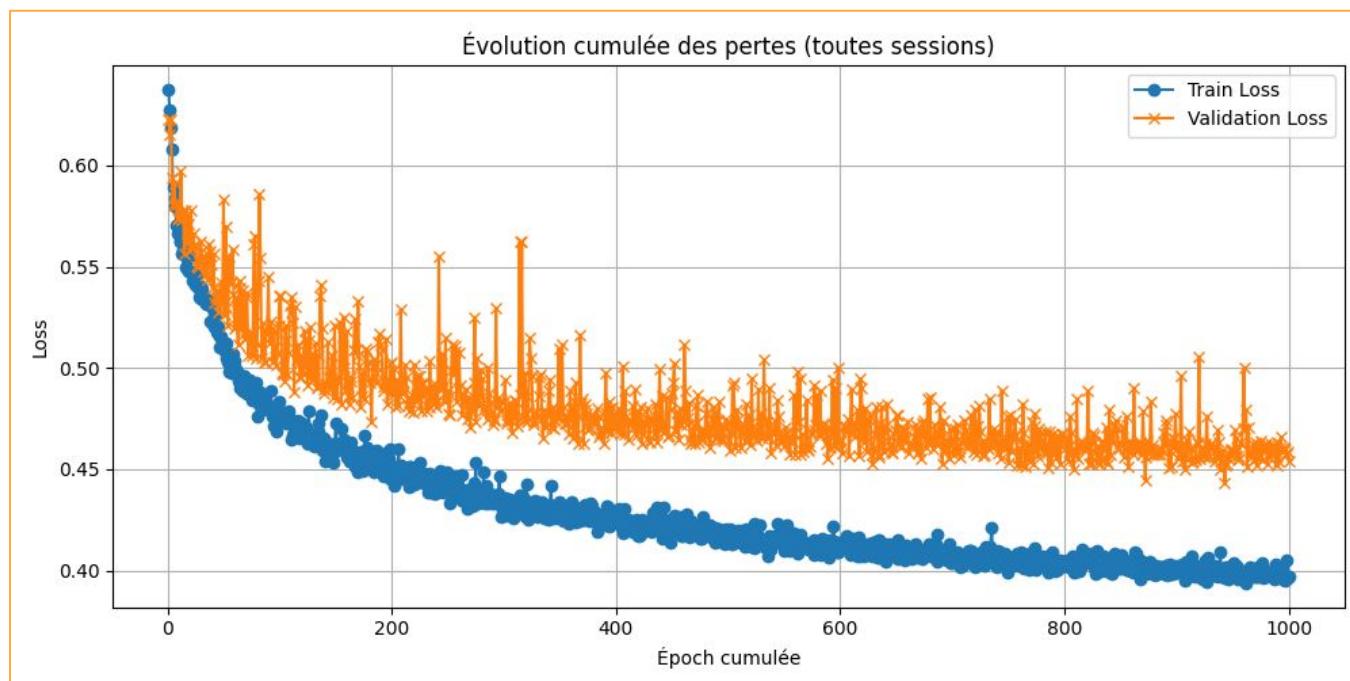
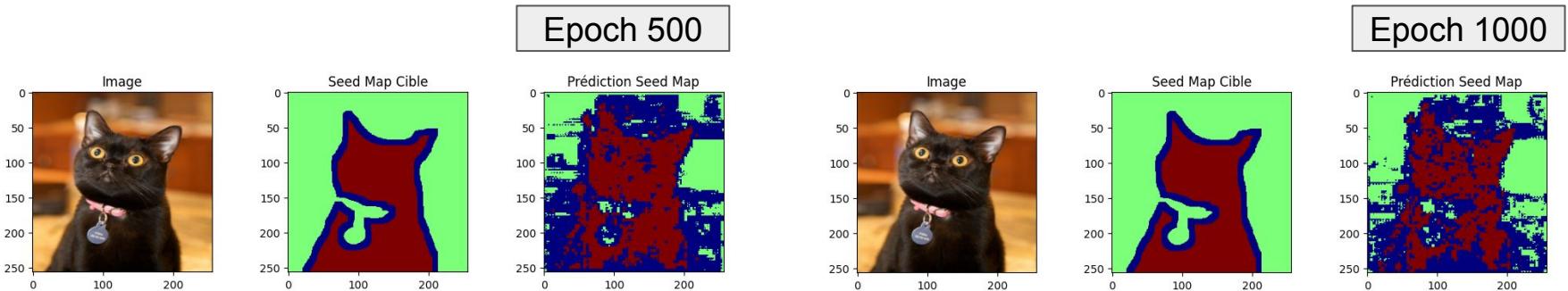
$$\mathcal{L}_{\text{sce}} = \frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(x_i)) + (1 - y_i) \cdot \log(1 - p(x_i))$$

y = résultat attendu, $p(y)$ = résultat prédit

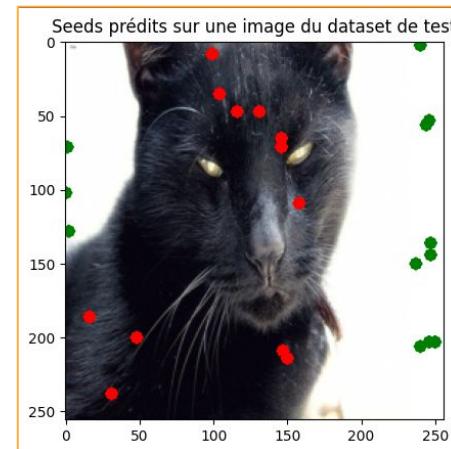
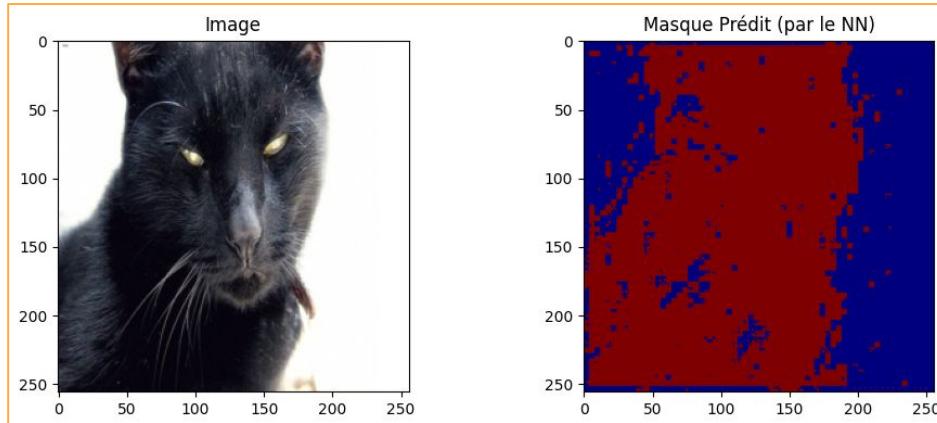


UMD - Visualizing
the Loss Landscape
of Neural Net

Progression de l'entraînement



Automatisation de la sélection des seeds : exemple de réussite



Sélecteur de couleur et affichage d'image

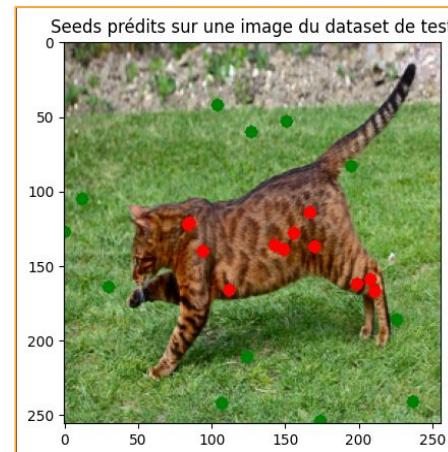
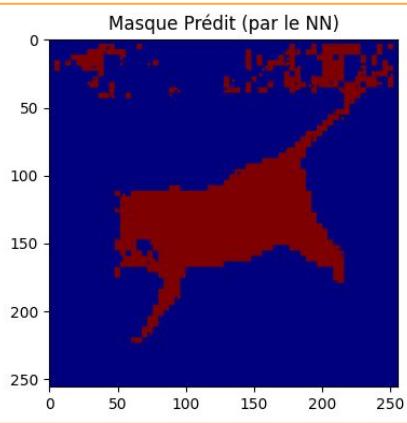
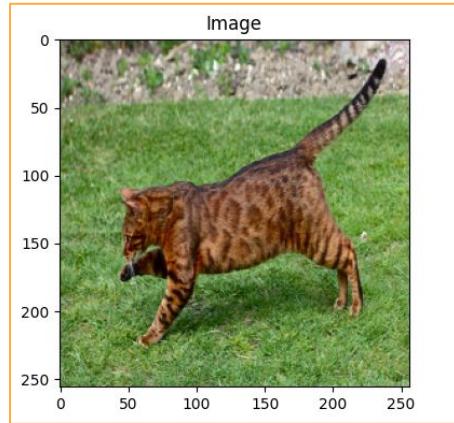
Choisir une image
Choisir couleur
Segmenter l'image
Sauvegarder preset
Charger preset
Reset Seeds

Aucune couleur sélectionnée

Preset 'cat60' chargé (661 points).
Segmenting image...
25%
50%
75%
100%
Nombre de marches convergentes : 64460
Image segmented in 19.2002 seconds



Automatisation de la sélection des seeds : exemple d'échec



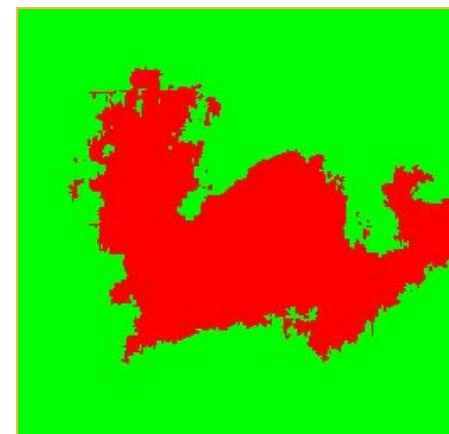
Sélecteur de couleur et affichage d'image

Choisir une image
Choisir couleur
Segmenter l'image
Sauvegarder preset
Charger preset
Reset Seeds

Aucune couleur sélectionnée

This screenshot shows a user interface for image segmentation. On the left, there is a sidebar with buttons for selecting an image, choosing a color, segmenting the image, saving a preset, loading a preset, and resetting seeds. Below these buttons is a message: "Aucune couleur sélectionnée" (No color selected). The main area displays the original image of the cat with several green and red circular seeds placed on it. To the right of the image is a smaller inset showing a zoomed-in view of the cat's head with a red crosshair over a seed.

Preset 'cat25' chargé (659 points).
Segmenting image...
25%
50%
75%
100%
Nombre de marches convergentes : 60463
Image segmented in 229.0188 seconds



Conclusion

- ❖ L'algorithme naïf s'est avéré très efficace après optimisation.
- ❖ L'algorithme utilisant les circuits électrique est difficile à mettre en place et les résultats ne sont pas forcément meilleurs que le naïf.
- ❖ En combinant le réseau de neurones et l'algorithme de segmentation par marches aléatoires on peut obtenir des résultats concluants pour des images “simples”.
- ❖ Piste d'amélioration : parallélisation des marches aléatoires.

MERCI POUR VOTRE ÉCOUTE !

Annexe : classe Graph

```
class Graph:
    def __init__(self):
        # Initialise le graphe avec une liste d'adjacence vide
        # Chaque clé sera un nœud, et la valeur associée sera une liste de tuples (voisin, poids)
        self.adjacency_list = {}

    def add_edge(self, node1, node2, weight):
        # Si node1 n'est pas encore dans le graphe, on l'ajoute avec une liste vide
        if node1 not in self.adjacency_list:
            self.adjacency_list[node1] = []

        # Idem pour node2
        if node2 not in self.adjacency_list:
            self.adjacency_list[node2] = []

        # On ajoute une arête entre node1 et node2 avec le poids donné
        # Puisque le graphe est non orienté, on ajoute l'arête dans les deux sens
        self.adjacency_list[node1].append((node2, weight))
        self.adjacency_list[node2].append((node1, weight))

    def get_neighbors(self, node):
        # Retourne la liste des voisins du nœud donné
        # Chaque voisin est un tuple (voisin, poids)
        return self.adjacency_list.get(node, [])

    def __str__(self):
        return str(self.adjacency_list)
```

Annexe : définition de la méthode d'initialisation de la classe *Image_Segmentation*

```
class Image_Segmentation:
    def __init__(self, seeds, image_path, height, width):
        self.img = Image.open(image_path)
        self.height = height
        self.width = width
        self.img = self.img.resize((self.width, self.height), Image.BICUBIC)
        self.img_array = np.array(self.img) # tableau de pixels
        self.graphe_image = Graph() # graphe des pixels
        self.marche_convergente = 0
        self.max_steps = 40000 # modifiable en fonction de la précision/rapidité escomptée
        self.sigma = 7 # modifiable en fonction de la précision/rapidité escomptée (peut aller à 3 pour moyenne_patch / peut aller à 8 pour RGB classique)
        self.seeds = seeds
        self.segmented_image = [[0 for _ in range(self.img.width)] for _ in range(self.img.height)]
        for seed in self.seeds:
            self.segmented_image[seed[0]][seed[1]] = self.seeds[seed]
        self.ajout_poids_graphe()
```

Annexe : méthodes de la classe *Image_Segmentation* pour l'attribution des poids du graphe (1/2)

```
def distance_couleur(self, p1, p2):
    # Récupère les couleurs RGB des deux pixels p1 et p2 dans l'image
    rgb1 = self.img_array[p1[0], p1[1]].astype(np.int16)
    rgb2 = self.img_array[p2[0], p2[1]].astype(np.int16)

    # Calcule la distance euclidienne entre les deux couleurs (dans l'espace RGB)
    distance = np.linalg.norm(np.array(rgb1) - np.array(rgb2))
    return distance

def poids_gauss(self, dist, sigma):
    # Calcule un poids selon une loi gaussienne en fonction de la distance
    # Utilisé pour donner plus d'importance aux pixels proches en couleur
    return exp(-(dist**2)/(sigma**2))

def poids_droite(self, p_origine):
    # Calcule le poids entre un pixel et son voisin de droite
    return self.poids_gauss(
        self.distance_couleur(p_origine, (p_origine[0], p_origine[1] + 1)),
        self.sigma
    )

def poids_dessous(self, p_origine):
    # Calcule le poids entre un pixel et son voisin du dessous
    return self.poids_gauss(
        self.distance_couleur(p_origine, (p_origine[0] + 1, p_origine[1])),
        self.sigma
    )
```

Annexe : méthodes de la classe Image_Segmentation pour l'attribution des poids du graphe (2/2)

```
def ajout_poids_graphe(self):
    for i in range(self.height):
        for j in range(self.width):
            # Si on n'est pas à la dernière colonne, relier le pixel (i, j) à son voisin de droite
            if j < (self.width - 1):
                self.graphe_image.add_edge(
                    (i, j), (i, j + 1), self.poids_droite((i, j)))
            )
            # Si on n'est pas à la dernière ligne, relier le pixel (i, j) à son voisin du dessous
            if i < (self.height - 1):
                self.graphe_image.add_edge(
                    (i, j), (i + 1, j), self.poids_dessous((i, j)))
            )
```

Annexe : méthode de la classe *Image_Segmentation* implémentant la marche aléatoire à partir d'un pixel

```
def marche_aleatoire(self, origin):
    # On commence la marche depuis le pixel 'origin'
    cur_pixel = origin

    # On récupère les voisins de ce pixel dans le graphe pondéré (graph_image)
    neighbors = self.graphe_image.get_neighbors(cur_pixel)

    i = 0 # Compteur d'étapes
    while i < self.max_steps:
        i += 1

        # Choix probabiliste d'un voisin :
        # Chaque voisin est choisi avec une probabilité proportionnelle au poids de l'arête
        cur_pixel = random.choices(
            population=[neighbor[0] for neighbor in neighbors], # liste des voisins
            weights=[neighbor[1] for neighbor in neighbors] # poids associés
        )[0]

        # Si on atteint un pixel déjà segmenté (ayant une étiquette non nulle)
        if self.segmented_image[cur_pixel[0]][cur_pixel[1]] != 0:
            self.marche_convergente += 1 # Compteur de marches ayant atteint une zone déjà classée

        # On récupère l'étiquette/couleur de ce pixel déjà étiqueté
        label = self.segmented_image[cur_pixel[0]][cur_pixel[1]]

        # On l'applique au pixel d'origine de la marche
        self.segmented_image[origin[0]][origin[1]] = label
        break # Marche terminée

        # Sinon, on continue la marche depuis ce nouveau pixel
        neighbors = self.graphe_image.get_neighbors(cur_pixel)

    # Si on atteint le nombre max d'étapes sans croiser une zone déjà segmentée :
    if i == self.max_steps:
        # On attribue au pixel 'origin' l'étiquette de la seed (graine) la plus proche géométriquement
        min_dist = float("inf")
        min_seed = None
        for seed in self.seeds:
            dist = sqrt((origin[0] - seed[0])**2 + (origin[1] - seed[1])**2)
            if dist < min_dist:
                min_dist = dist
                min_seed = seed

        # On applique l'étiquette de cette seed au pixel
        self.segmented_image[origin[0]][origin[1]] = self.seeds[min_seed]
```

Annexe : méthode de *Image_Segmentaion* appliquant la marche aléatoire sur chaque pixel non encore marqués

```
def color_all_pixels(self):
    all_pixels = [(i,j) for i in range(self.img.height) for j in range(self.img.width)] # Liste avec toutes les coordonnées de pixel possibles
    random.shuffle(all_pixels)
    k = 0
    for (i, j) in all_pixels:
        k += 1
        if k == self.img.height*self.img.width//4:
            print("25%")
        elif k == self.img.height*self.img.width//2:
            print("50%")
        elif k == 3*self.img.height*self.img.width//4:
            print("75%")
        if (i, j) not in self.seeds:
            self.marche_aleatoire((i, j))
    print("100%")
    print(f"Nombre de marches convergentes : {self.marche_convergente}")
```

Annexe : Train loop (1/2)

```
def train(model, train_loader, val_loader, device='cuda', epochs=5, lr=1e-3):
    # Envoi du modèle sur le GPU ou CPU
    model = model.to(device)

    # Définition des poids pour chaque classe (ex : objet vs fond), utile en cas de déséquilibre
    weights = torch.tensor([1.0, 5.0]).to(device)
    criterion = nn.CrossEntropyLoss(ignore_index=-1, weight=weights) # -1 = pixels ignorés
    optimizer = torch.optim.Adam(model.parameters(), lr=lr) # Optimiseur Adam

    # Listes pour stocker les pertes par époque
    train_losses = []
    val_losses = []

    for epoch in range(epochs):
        model.train() # Mode entraînement
        epoch_loss = 0.0

        # Boucle sur les batches d'entraînement
        for images, seed_maps, _ in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs}"):
            # Passage des données sur le device
            images = images.to(device)
            seed_maps = seed_maps.to(device)

            # Prédiction du modèle
            outputs = model(images)

            # Calcul de la perte (comparaison prédictions vs étiquettes)
            loss = criterion(outputs, seed_maps)

            # Rétropropagation du gradient et mise à jour des poids
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # Accumulation de la perte sur l'époque
            epoch_loss += loss.item()

        # Moyenne des pertes d'entraînement sur l'époque
        avg_train_loss = epoch_loss / len(train_loader)
        train_losses.append(avg_train_loss)

    # Sauvegarde du modèle
    torch.save(model.state_dict(), 'model.pth')
```

Annexe : Train loop (2/2)

```
# -----
# Phase d'évaluation sur validation
model.eval() # Mode évaluation (désactive dropout, batchnorm)
val_loss = 0.0
with torch.no_grad(): # Pas de calcul de gradient ici
    for images, seed_maps, _ in val_loader:
        images = images.to(device)
        seed_maps = seed_maps.to(device)

        outputs = model(images)
        loss = criterion(outputs, seed_maps)
        val_loss += loss.item()

# Moyenne des pertes de validation sur L'époque
avg_val_loss = val_loss / len(val_loader)
val_losses.append(avg_val_loss)

# Affichage du résumé de L'époque
print(f"Epoch {epoch+1} - Train Loss: {avg_train_loss:.4f} | Val Loss: {avg_val_loss:.4f}")

# Sauvegarde des courbes de pertes dans un fichier CSV
save_losses_to_csv(train_losses, val_losses)
```

Annexe : fonction permettant d'extraire des seeds de la seed map

```
def get_limited_seeds(fg_probs, bg_probs, n_fg=3, n_bg=3, radius=3, fg_thresh=0.9, bg_thresh=0.9):
    """
    Crée un dictionnaire de seeds[(y,x)] = label à partir des probabilités de foreground/background.
    - n_fg : nombre de seeds foreground
    - n_bg : nombre de seeds background
    """
    H, W = fg_probs.shape
    seeds = {}

    # 1. Masques binaires
    fg_mask = fg_probs > fg_thresh
    bg_mask = bg_probs > bg_thresh

    # 2. Coordonnées des candidats
    fg_coords = list(zip(*np.where(fg_mask)))
    bg_coords = list(zip(*np.where(bg_mask)))

    # 3. Sélection aléatoire de n points
    if len(fg_coords) >= n_fg:
        fg_sample = random.sample(fg_coords, n_fg)
    else:
        fg_sample = fg_coords

    if len(bg_coords) >= n_bg:
        bg_sample = random.sample(bg_coords, n_bg)
    else:
        bg_sample = bg_coords

    # 4. Remplir le dictionnaire avec des disques
    for y, x in fg_sample:
        ys, xs = draw_disk(y, x, radius, (H, W))
        for yy, xx in zip(ys, xs):
            seeds[(yy, xx)] = "#ff0000"

    for y, x in bg_sample:
        ys, xs = draw_disk(y, x, radius, (H, W))
        for yy, xx in zip(ys, xs):
            seeds[(yy, xx)] = "#00ff00"

    return seeds
```

Annexe : fonction d'affichage des prédictions du CNN

```
def visualize_predictions_2(model, dataloader, device='cuda'):
    model.eval()
    model = model.to(device)
    batch = 0 # compteur de batch
    w_batch = 12 # index du batch à afficher
    with torch.no_grad():
        for images, seed_maps, _ in dataloader:
            if batch == w_batch:
                images = images.to(device)
                seed_maps = seed_maps.to(device)

                outputs = model(images) # [B, 2, H, W]
                probs = F.softmax(outputs, dim=1)

                fg_probs = probs[:, 1] # foreground prob
                bg_probs = probs[:, 0] # background prob

                # Seuiller les *seeds* prédits (proba > 0.85)
                pred_seed_map = torch.full_like(seed_maps, -1)
                pred_seed_map[fg_probs > 0.85] = 1
                pred_seed_map[bg_probs > 0.85] = 0

                # Afficher les 4 premiers exemples
                for i in range(4):
                    plt.figure(figsize=(12, 3))

                    plt.subplot(1, 3, 1)
                    plt.imshow(images[i].cpu().permute(1, 2, 0))
                    plt.title("Image")

                    plt.subplot(1, 3, 2)
                    plt.imshow(seed_maps[i].cpu(), cmap='jet', vmin=-1, vmax=1)
                    plt.title("Seed Map Cible")

                    plt.subplot(1, 3, 3)
                    plt.imshow(pred_seed_map[i].cpu(), cmap='jet', vmin=-1, vmax=1)
                    plt.title("Prédiction Seed Map")

                    plt.show()
                break # afficher juste un batch
            batch +=1
```

Annexe : méthode principale de la classe UI créant l'interface graphique (1/2)

```
def run_ui(self):
    # Frame de gauche (boutons)
    self.color_frame = tk.Frame(self.root)
    self.color_frame.pack(side="left", fill="y", padx=5, pady=5)

    # Bouton pour choisir l'image
    self.choose_image_button = tk.Button(
        self.color_frame, text="Choisir une image", command=self.choose_image
    )
    self.choose_image_button.pack(pady=5)

    # Bouton choisir couleur
    self.choose_color_button = tk.Button(self.color_frame, text="Choisir couleur",
                                         command=self.create_color_palette)
    self.choose_color_button.pack(pady=5)

    # Bouton segmenter
    self.segment_button = tk.Button(self.color_frame, text="Segmenter l'image",
                                    command=self.segmentImage)
    self.segment_button.pack(pady=5)

    # Bouton sauver
    self.save_button = tk.Button(self.color_frame, text="Sauvegarder preset",
                                command=self.show_save_preset_window)
    self.save_button.pack(pady=5)

    # Bouton charger
    self.load_button = tk.Button(self.color_frame, text="Charger preset",
                                command=self.show_load_preset_window)
    self.load_button.pack(pady=5)
```

Annexe : méthode principale de la classe UI créant l'interface graphique (2/2)

```
# Bouton Reset Seeds
self.reset_button = tk.Button(
    self.color_frame, text="Reset Seeds", command=self.reset_seeds
)
self.reset_button.pack(pady=5)

# Label couleur sélectionnée
self.color_label = tk.Label(self.color_frame, text="Aucune couleur sélectionnée")
self.color_label.pack(pady=5)

# Canvas pour l'image
self.image_canvas = tk.Canvas(self.root, width=self.width, height=self.height)
self.image_canvas.pack(side="left")

# Charger l'image
img = Image.open(self.image_path).resize((self.width, self.height), Image.BICUBIC)
self.display_img = img # Conserver l'objet PIL
self.photo = ImageTk.PhotoImage(img)
self.image_canvas.create_image(0, 0, anchor="nw", image=self.photo)

# Clic sur l'image
self.image_canvas.bind("<Button-1>", self.on_click)

# Canvas Loupe
self.magnifier_canvas = tk.Canvas(self.root,
                                   width=self.magnifier_size * self.magnifier_scale,
                                   height=self.magnifier_size * self.magnifier_scale)
self.magnifier_canvas.pack(side="left", padx=5)

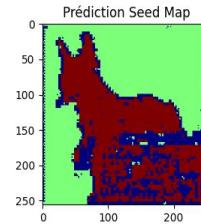
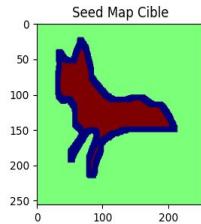
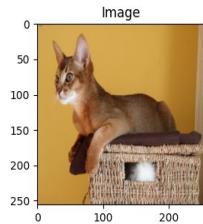
# Mouvement souris = Loupe
self.image_canvas.bind("<Motion>", self.show_magnifier)
```

Annexe : approche électrique code

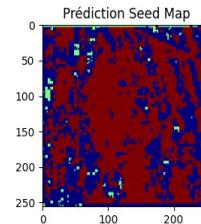
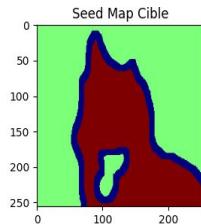
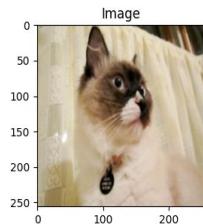
```
def build_linear_algebra(self):  
    print('building graph')  
    self.build_weighted_graph()  
    self.ordered_nodes = get_ordered_nodelist(  
        list(self.graph), list(self.seeds_dic.keys()))  
    print('computing laplacian')  
    self.laplacian = networkx.laplacian_matrix(  
        self.graph, nodelist=self.ordered_nodes, weight='weight')  
    print('extracting sub-matrices')  
    self.laplacian_unseeded = self.laplacian[self.K:, self.K:]  
    self.b_transpose = self.laplacian[self.K:, :self.K]  
  
def solve_linear_systems(self):  
    print('solving linear systems')  
    unseeded_potentials_list = []  
    for seed_index in range(self.K):  
        print(f'Solving system {seed_index+1} out of {self.K}')  
        seeds_vector = [0] * self.K  
        seeds_vector[seed_index] = 1  
        unseeded_potentials = scipy.sparse.linalg.spsolve(  
            self.laplacian_unseeded, -self.b_transpose @ seeds_vector)  
        unseeded_potentials_list.append(unseeded_potentials)  
    return unseeded_potentials_list  
  
def assign_max_likelihood(self, unseeded_potentials_list):  
    print('Assigning maximum likelihood seed')  
    for pixel_index in range(self.K, self.pixel_number):  
        pixel_coords = self.ordered_nodes[pixel_index]  
        pixel_probabilities = [potentials[pixel_index - self.K]  
            for potentials in unseeded_potentials_list]  
        argmax_seed_index = np.argmax(pixel_probabilities)  
        argmax_seed_coords = list(self.seeds_dic.keys())[argmax_seed_index]  
        self.pixel_colour_dic.update({  
            pixel_coords: self.seeds_dic[argmax_seed_coords]  
        })
```

<https://github.com/MB-29/Random-Walker-Image-Segmentation>

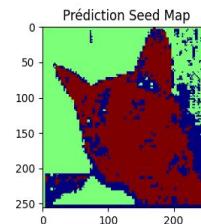
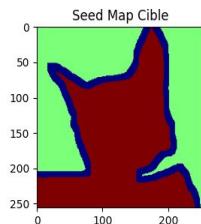
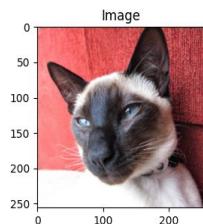
Annexe : quelques cas intéressants de seedmap prédites par CNN



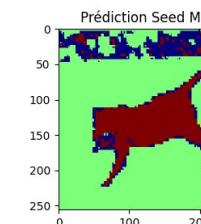
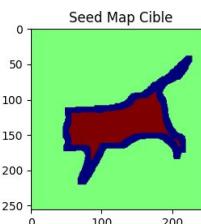
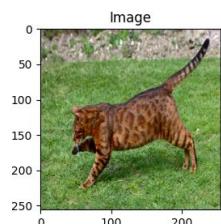
Epoch 800



Epoch 700



Epoch 850



Epoch 500