

# Compression d'images médicales avec des réseaux de neurones artificiels

Anouar ACHGHAF  
Numéro de dossier : 19089

## Table des matières

<b>I</b>	<b>Introduction</b>	<b>2</b>
<b>II</b>	<b>Le format JPEG</b>	<b>2</b>
<b>1</b>	<b>Sous-échantillonnage</b>	<b>2</b>
<b>2</b>	<b>Transformée en cosinus discrète et quantification</b>	<b>3</b>
2.1	Transformée en cosinus discrète . . . . .	3
2.2	Quantification . . . . .	4
<b>3</b>	<b>Compression sans pertes</b>	<b>4</b>
<b>III</b>	<b>Carte auto-organisatrice</b>	<b>4</b>
<b>4</b>	<b>Heuristique</b>	<b>5</b>
<b>5</b>	<b>Structure du réseau</b>	<b>5</b>
5.1	Structure générale . . . . .	5
5.2	En imagerie . . . . .	6
<b>6</b>	<b>Algorithme d'apprentissage</b>	<b>6</b>
6.1	L'algorithme . . . . .	6
6.2	Analyse de complexité . . . . .	6
<b>7</b>	<b>Amélioration de l'algorithme de recherche</b>	<b>6</b>
7.1	Une première approximation . . . . .	6
7.2	Une deuxième approximation . . . . .	7
<b>8</b>	<b>Résultats</b>	<b>7</b>
8.1	Estimation des résultats . . . . .	7
8.2	Résultats avec la méthode exacte . . . . .	7
8.3	Première méthode d'approximation . . . . .	8
8.4	Deuxième méthode d'approximation . . . . .	9
8.5	Comparaison de temps d'exécution . . . . .	11
	<b>Références bibliographiques</b>	<b>11</b>
<b>IV</b>	<b>Annexes</b>	<b>12</b>

# Première partie

## Introduction

Les hôpitaux produisent généralement des dizaines de gigaoctets de données par jour. Les grands hôpitaux, comme l'hôpital universitaire de Vienne, produisent actuellement plus de 100 GO d'images par jour. Ils ont aussi l'obligation légale d'archiver ces données pendant une certaine période.

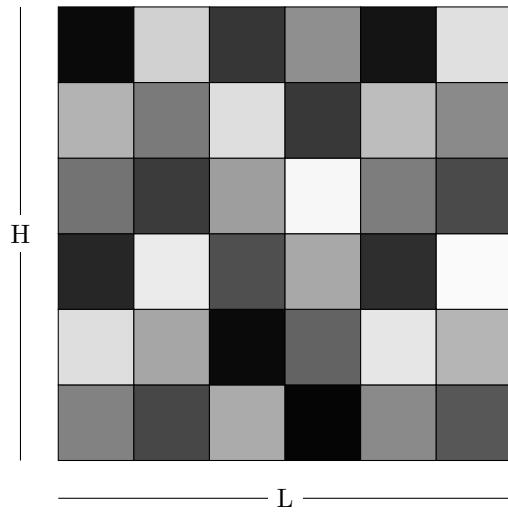
Pour cette raison, la problématique d'optimisation des algorithmes de compression d'image est pertinente.

Actuellement, le standard international pour le stockage d'images médicales, à savoir le standard DICOM, ne supporte que des formats d'images très généralistes (PNG, JPEG, etc...), et de fait que des formats de compressions généralistes. On s'intéresse donc à la possibilité de tirer parti des spécificités des images médicales pour concevoir des algorithmes spécialisés potentiellement plus efficaces, notamment à l'aide de réseaux de neurones pour effectuer de la compression avec perte.

# Deuxième partie

## Le format JPEG

Dans cette partie, on s'intéresse à la compression JPEG [3] d'images en noir et blanc. On se donne une image de dimensions  $H \times L$ , représentée par la matrice  $D \in \mathcal{M}_{H,L}(\mathbf{R}^+)$ , où  $D_{i,j}$  ( $0 \leq D_{i,j} \leq 255$ ) représente la luminance du pixel  $(i,j)$ .



On commence par partitionner l'image en blocs de dimensions  $8 \times 8$ . Pour simplifier les représentations, on utilisera des blocs de dimensions  $6 \times 6$ .

La compression JPEG se fait en plusieurs étapes.

### 1 Sous-échantillonnage

Le sous-échantillonnage consiste à réduire le nombre d'échantillons à traiter, donc de diminuer le volume de l'image numérique, en remplaçant la luminance de deux pixels adjacents par une seule valeur, et réduire ainsi la taille du fichier d'un facteur  $\approx \frac{1}{2}$ . Cette étape n'est pas systématiquement utilisée pour la luminance. En revanche, elle est utilisée pour la chrominance ; cela est dû au fait que l'œil humain distingue avec plus de précision la différence de luminance.

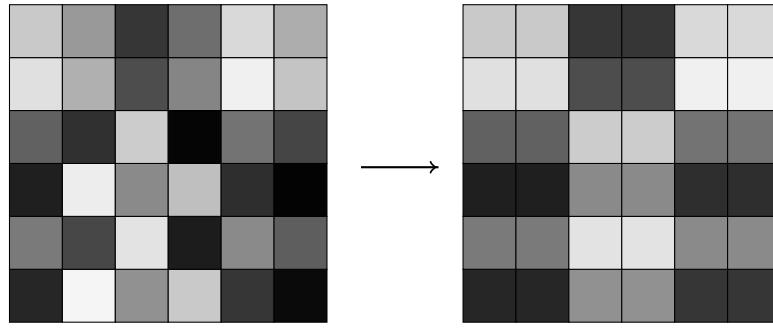


FIGURE 1 – Exemple de sous-échantillonnage

## 2 Transformée en cosinus discrète et quantification

### 2.1 Transformée en cosinus discrète

La transformée en cosinus discrète est une transformation similaire à la transformée de Fourier discrète, proposée par Nasir Ahmed en 1972 [1], et souvent utilisée dans la compression d'images.

On considère la matrice :

$$P_{i,j} = \begin{cases} \frac{1}{\sqrt{8}}, & \text{si } i = 0 \\ \sqrt{\frac{2}{8}} \cos \frac{(2j+1)i\pi}{16}, & \text{sinon} \end{cases}$$

Alors la matrice  $PDP^\top$  correspond aux degrés de contribution des composantes en cosinus, i.e.  $(PDP^\top)_{i,j}$  est l'énergie de la case de coordonnées  $(i, j)$  des composantes :

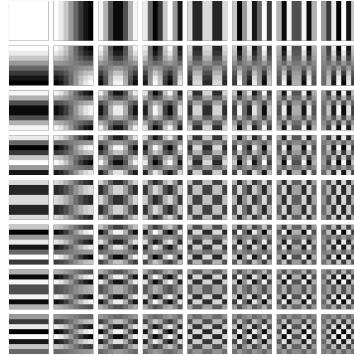


FIGURE 2 – Composantes en cosinus

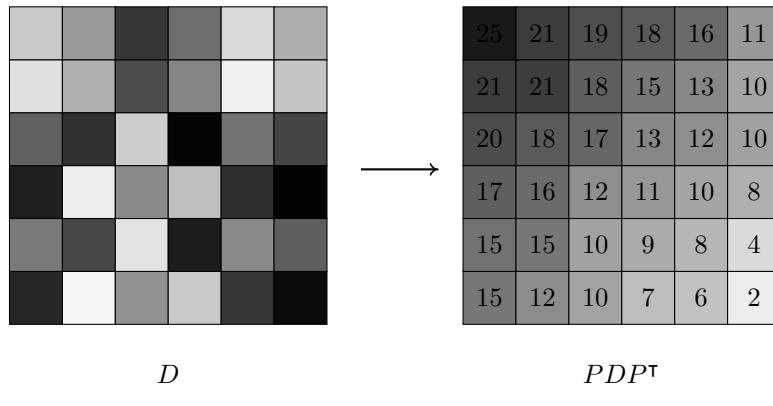


FIGURE 3 – Application de la TCD

Cette multiplication n'implique aucune perte du fait que la matrice  $P$  est orthogonale.

## 2.2 Quantification

L'étape de quantification consiste à supprimer les composantes dont l'énergie est inférieure à une certaine valeur dépendant du taux de compression.

25	21	19	18	16	11
21	21	18	15	13	10
20	18	17	13	12	10
17	16	12	11	10	8
15	15	10	9	8	4
15	12	10	7	6	2

25	21	19	18	16	0
21	21	18	15	13	0
20	18	17	13	0	0
17	16	0	0	0	0
15	15	0	0	0	0
15	0	0	0	0	0

FIGURE 4 – Quantification

Les hautes fréquences, étant réservées à des changements rapides d'intensité des pixels, sont en général minimes dans une image donc plus susceptibles d'être supprimées, sans un grand impact sur la qualité de l'image.

## 3 Compression sans pertes

Lors de la sauvegarde des données produites par les étapes précédentes, on utilise plusieurs algorithmes de compression sans perte :

1. On commence par ordonner les coefficients de la matrice quantifiée en zigzags comme le montre la figure 5, on obtient ainsi la séquence  $(26, -3, 0, -3, \dots, 0)$ .
2. Codage différentiel : on transforme la séquence obtenue par la séquence des différences entre éléments successifs ; on obtient donc la séquence  $(26, 23, 3, -3, \dots, 0)$ . L'intérêt de cette transformation est d'obtenir des éléments généralement plus petits e.g.  $(0, 15, 30, 45) \rightarrow (0, 15, 15, 15)$ .
3. Codage par plages : on remplace la succession d'une même valeur par un couple indiquant la valeur et son nombre d'occurrences. On obtient ainsi  $((1, -26), (1, 23), \dots, (1, 1), (37, 0))$ .
4. Codage de Huffman et mise en mémoire tampon de la sortie : des méthodes optimisant l'encodage binaire des données, n'entrant pas dans le champ de la présente étude.

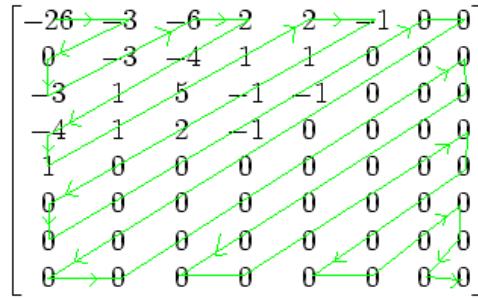


FIGURE 5 – Ordénement en zigzags

# Troisième partie

## Carte auto-organisatrice

### 4 Heuristique

On considère  $p$  points  $a_1, \dots, a_p$  dans  $\mathbb{R}^2$ .

On définit la distance entre 2 points  $A$  et  $B$  comme la distance euclidienne entre eux :  $\text{dist}(A, B) = \|\mathbf{AB}\|$ .

Le but est de représenter l'ensemble des  $p$  points par  $q$  points  $b_1, \dots, b_q$  en minimisant la valeur de :

$$\sum_{i=1}^p \min_{j \in \{1, \dots, q\}} \text{dist}(a_i, b_j) \quad (1)$$

En guise d'exemple, on considère  $p = 10$  points  $a_1, \dots, a_p$  du plan représentés par les points noirs dans la figure 6. Le but est de discréteriser le plan en le divisant en  $q = 3$  régions, dont tout point s'y trouvant  $a_i$  est représenté par le point  $b_j$  correspondant. On commence par générer aléatoirement les 3 points  $b_1, b_2$  et  $b_3$ , représentés par les points rouges, puis on les repositionne dans des endroits judicieux à travers l'algorithme d'entraînement de manière à minimiser la valeur de (1). Tout point introduit ensuite dans le plan aura comme représentant le point  $b_i$  se trouvant dans sa région.

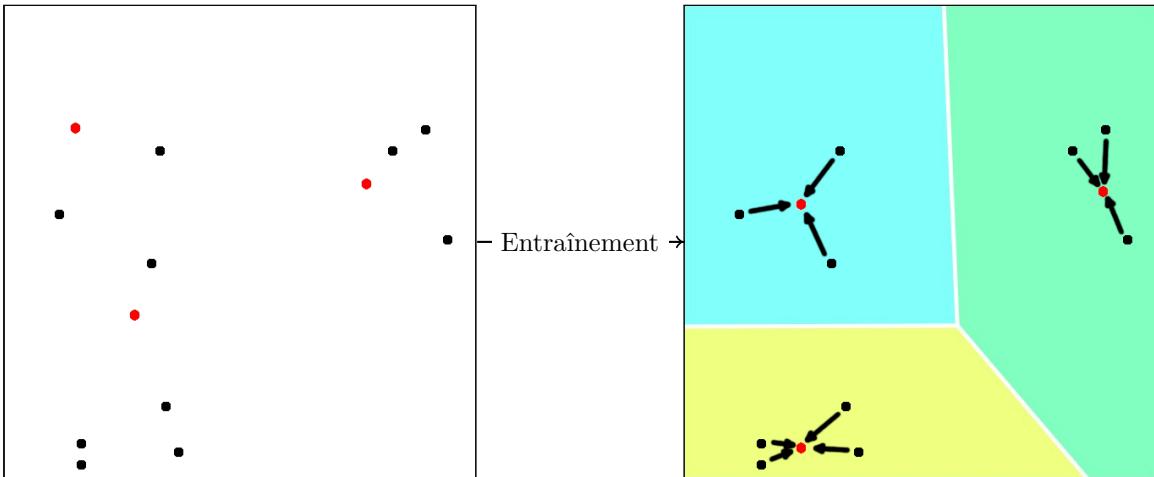


FIGURE 6 – Exemple dans  $\mathbb{R}^2$

### 5 Structure du réseau

#### 5.1 Structure générale

Proposée par Teuvo Kohonen en 1989 [2], l'architecture de la carte auto-organisatrice est définie par une couche d'entrée et une couche de sortie seulement. Les neurones d'entrée correspondent aux coordonnées du vecteur donné en entrée  $a$ , alors que chacun des neurones de sortie correspond à un vecteur  $b_i$  de coordonnées  $w_{1,i}, \dots, w_{n,i}$ .

Plus formellement :

$$a = (x_1, \dots, x_n) \quad (2)$$

$$b_i = (w_{1,i}, w_{2,i}, \dots, w_{n,i}), \forall i \in \{1, \dots, m^2\} \quad (3)$$

Chacun des neurones d'entrée  $x_i$  est lié à tous les vecteurs de sortie par une arête de poids  $w_{i,j}$ , ( $j \in \{1, \dots, m^2\}$ ), et la couche de sortie est organisée sous forme d'une carte où chaque neurone est lié aux neurones adjacents (voir figure 7).

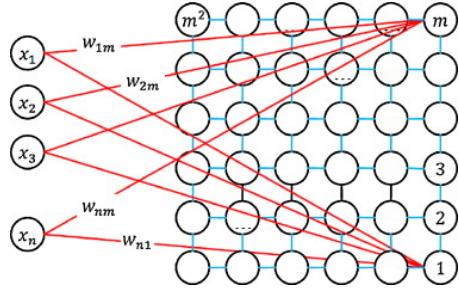


FIGURE 7 – Structure du réseau de la carte auto-organisatrice

## 5.2 En imagerie

Dans notre cas, la couche d'entrée est constituée de  $n \times n$  neurones correspondant aux luminances des pixels d'un bloc de dimensions  $n \times n$ , le neurone  $i$  de la couche de sortie correspond à un bloc de pixels de dimensions  $n \times n$  de luminances  $w_{1,i}, \dots, w_{n^2,i}$ .

# 6 Algorithme d'apprentissage

## 6.1 L'algorithme

On fixe la taille de la couche de sortie à  $m \times m$ , et la taille de la couche d'entrée à  $n \times n$ . L'apprentissage se déroule comme suit :

1. On génère les  $m^2$  points de la couche de sortie aléatoirement de  $\mathbb{R}^{n^2}$
2. On choisit une fonction décroissante  $\mu$  de  $\mathbb{R}^+$  dans  $[0, 1]$ , au but d'avoir une meilleure convergence
3. On choisit un bloc aléatoire de  $n \times n$  pixels de l'ensemble des données d'entraînement
4. On cherche le neurone de sortie dont la distance euclidienne au bloc d'entrée est minimale
5. On rapproche ce neurone et ses voisins du point d'entrée selon la formule :

$$w_{i,j} := w_{i,j} + \mu(t) \cdot (x_i - w_{i,j})$$

6. On reprend à l'étape 3 jusqu'à convergence

## 6.2 Analyse de complexité

Que ce soit pour l'entraînement ou pour le calcul de la sortie, le problème est de trouver le point le plus proche du bloc d'entrée. En calculant la distance euclidienne de chaque neurone de la couche de sortie du bloc d'entrée, on obtient une complexité temporelle de  $\mathcal{O}(n^2 m^2)$ .

Pour une image de dimensions  $H \times L$ , on traite  $\approx \frac{HL}{n^2}$  blocs, pour une complexité totale de  $\mathcal{O}(m^2 HL)$ .

# 7 Amélioration de l'algorithme de recherche

Pour les vecteurs dans un espace de dimensions  $n$ , l'algorithme naïf de recherche du point le plus proche devient trop lent. Les algorithmes de partitionnement de l'espace étant hors de portée dans notre contexte<sup>1</sup>, on se contentera d'algorithmes approximatifs.

## 7.1 Une première approximation

Un algorithme trivial de recherche du point le plus proche consiste à chercher parmi un nombre réduit de points afin d'obtenir des résultats acceptables tout en gardant un temps d'exécution raisonnable. On va choisir aléatoirement  $m$  points parmi les  $m^2$  points, pour une complexité temporelle finale de  $\mathcal{O}(mHL)$ . Cependant, on s'attend à voir plus de bruit sur les images produites.

1. Il existe des algorithmes efficaces de recherche lorsque la dimension  $n$  est petite (inférieure à 15). La structure la plus connue est l'arbre k-d, introduite en 1975. Un problème majeur de ces méthodes est de souffrir de la malédiction de la dimension : lorsque la dimension  $n$  est trop grande, ces méthodes ont des performances comparables ou inférieures à une recherche linéaire.

## 7.2 Une deuxième approximation

Le deuxième algorithme qu'on a exploré consiste à trier les neurones de la couche de sortie selon leur norme. Puisqu'il s'agit du calcul de la sortie, les coordonnées des points de la couche de sortie ne subissent aucun changement. De ce fait, on ne trie la liste de neurones qu'une seule fois. Ensuite, on cherche le neurone le plus proche parmi un nombre constant de neurones dont la distance au bloc d'entrée est minimale. En effectuant une dichotomie lors de la recherche du bloc de plus proche norme, on obtient une complexité temporelle finale  $\mathcal{O}(HL \log m)$ .

On s'attend à la possibilité d'avoir de grosses erreurs pour de grandes valeurs de  $n$  et  $m$ , correspondant à la situation de la figure 8 en dimension 3 : en cherchant les vecteurs proches de  $s_1$ , on favorise le choix de vecteurs de normes proches de la norme de  $s_1$  sans prendre en compte leurs distances de  $s_1$ , on favorise donc le choix du vecteur  $s_2$ . Néanmoins, les résultats obtenus montrent que cette situation est peu probable.

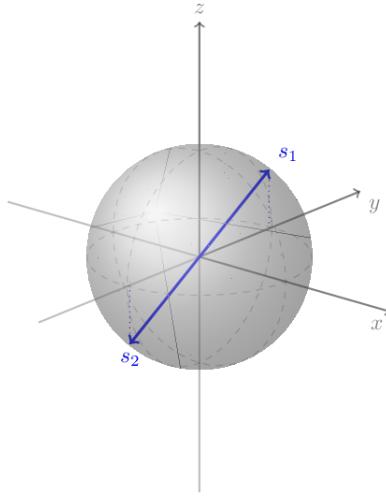


FIGURE 8 – Exemple de situation d'échec de l'algorithme

## 8 Résultats

### 8.1 Estimation des résultats

Pour estimer la taille du fichier produit par la méthode proposée, on considère un ensemble de représentants fixe composé de  $m^2$  points. Pour chaque bloc  $n \times n$ , on sauvegarde une seule valeur dans  $\{0, \dots, m^2 - 1\}$ , le représentant du bloc.

Pour  $m^2 < 2^k$ , on peut utiliser  $k$  bits pour chaque bloc, pour une taille totale de  $k \cdot \frac{HL}{n^2}$  bits au lieu de  $c \cdot HL$  bits ( $c \in \mathbb{R}^+$ ) avec une représentation naïve.

### 8.2 Résultats avec la méthode exacte

Comme le montre la figure 9, l'erreur produite avec la carte auto-organisatrice décroît de manière régulière en fonction de  $m$  (le nombre de représentants).

La courbe 9a de la figure 9 montre que la méthode proposée a beaucoup de potentiel pour de faibles coefficients de réduction, surtout en prenant en compte la décroissance apparente de l'erreur en fonction de  $m$ , tout en gardant la taille du fichier produit fixe.

Pour des tailles de fichiers comparables (fig. 10), l'image produite à l'aide de la carte auto-organisatrice est de bien meilleure qualité ; il est tout à fait raisonnable de considérer un tel taux de compression avec la carte auto-organisatrice vu que la différence est presque invisible à l'œil nu.

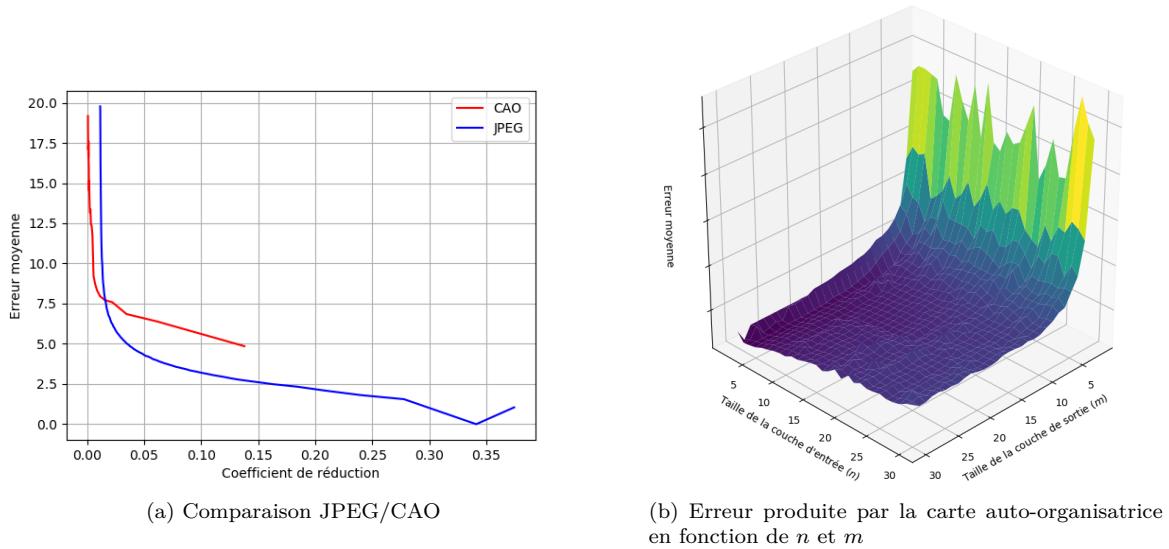


FIGURE 9 – Résultats obtenus avec la méthode exacte

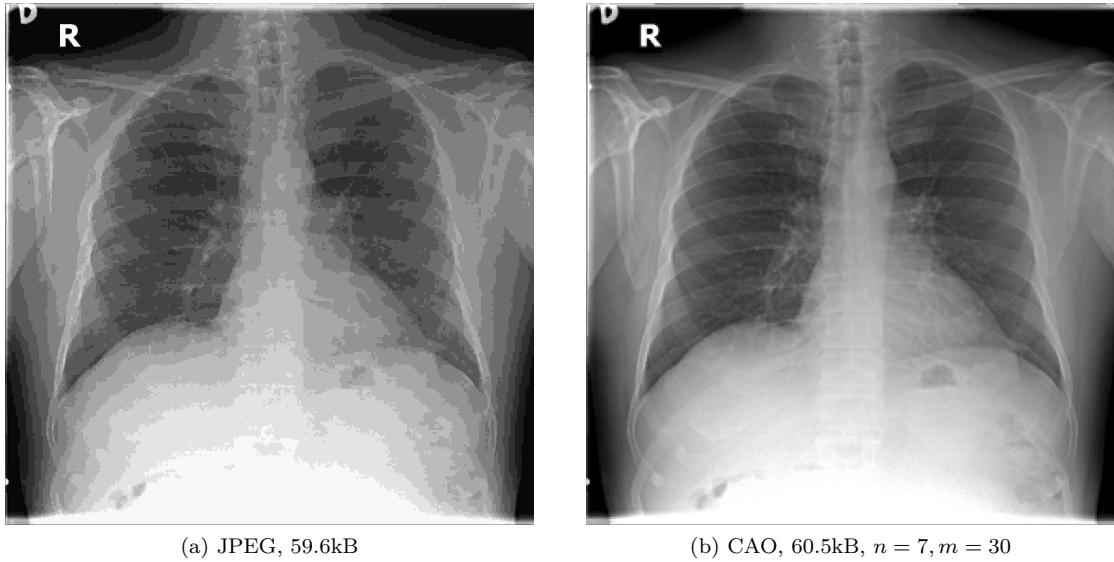


FIGURE 10 – Résultats obtenus avec la méthode exacte

### 8.3 Première méthode d'approximation

La figure 11 montre que la première méthode d'approximation est beaucoup moins régulière, et produit beaucoup plus d'erreurs et de bruit dans les images produites (fig. 12), résultat auquel on s'attendait.

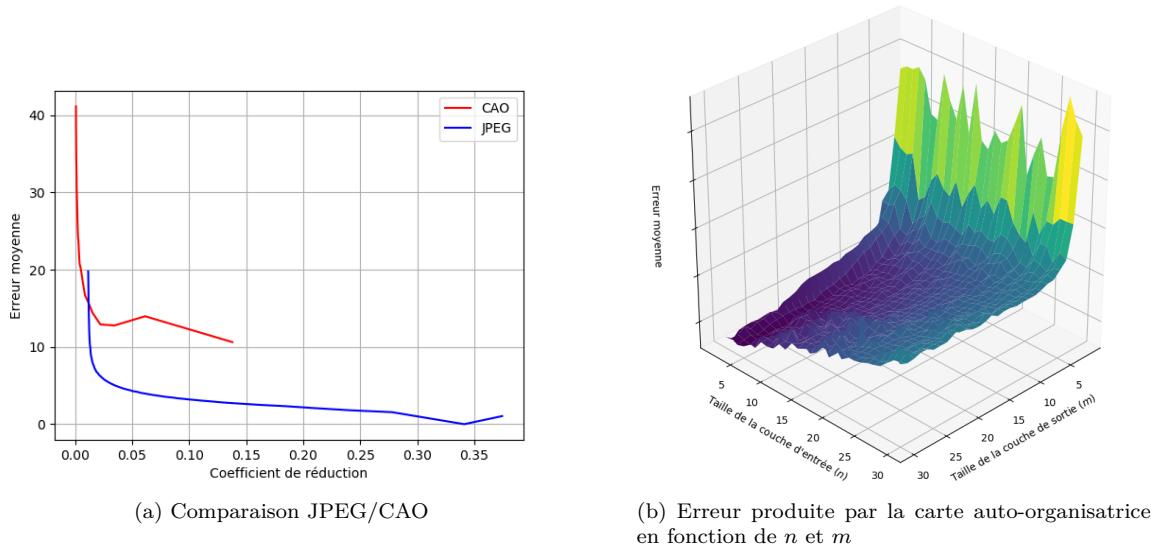


FIGURE 11 – Résultats obtenus avec la première méthode d’approximation

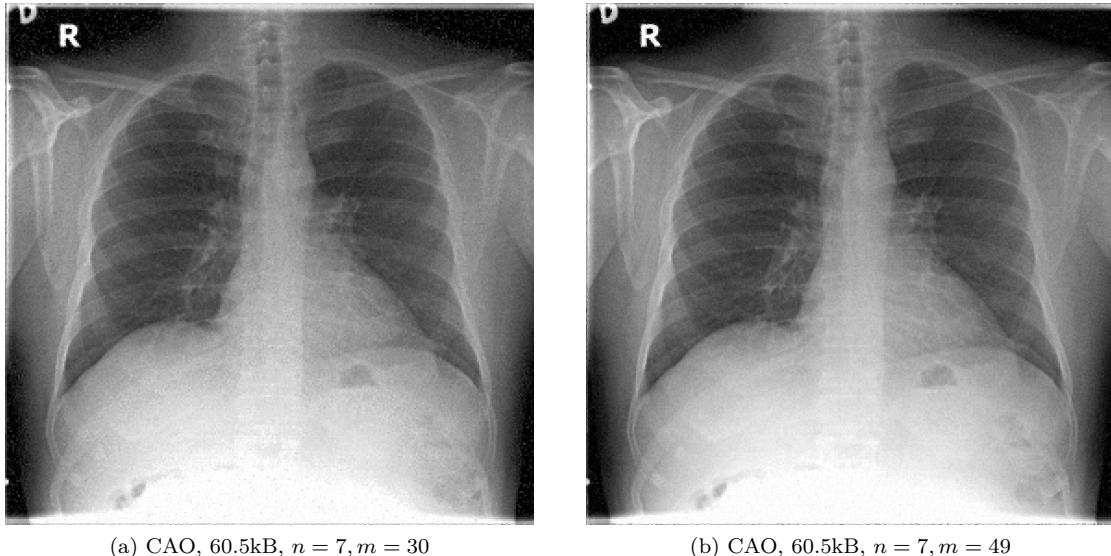
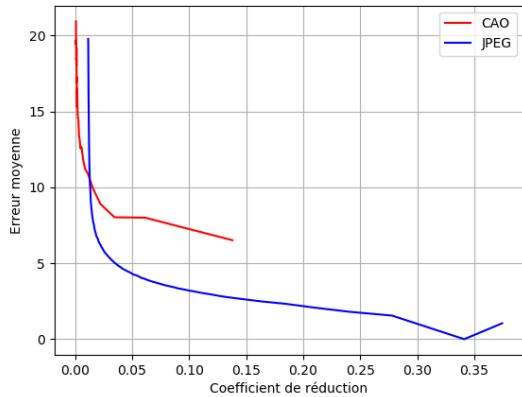


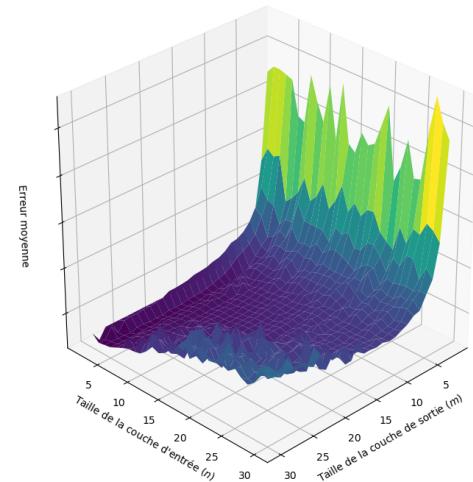
FIGURE 12 – Images produites par la première méthode d’approximation

#### 8.4 Deuxième méthode d’approximation

La figure 13 montre que la deuxième méthode d’approximation produit des images beaucoup plus précises que celles produites par la première méthode, la courbe 13a est très similaire à celle de la méthode exacte, il est donc plus intéressant d’utiliser cette méthode. Cependant, on tombe sur la situation décrite en 7.2 pour de grandes valeurs de  $m$ , ce qui est à l’origine des artefacts apparaissant dans la figure 15.

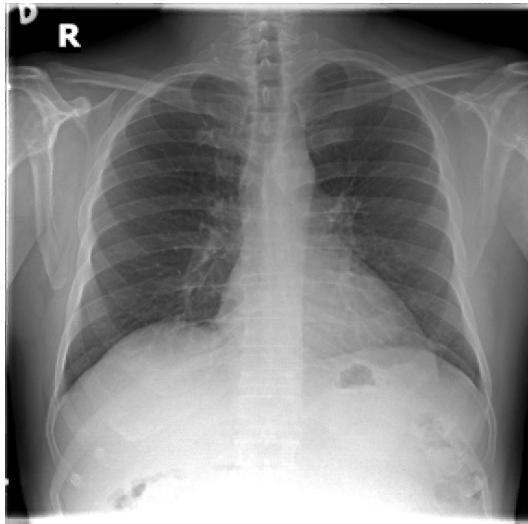


(a) Comparaison JPEG/CAO

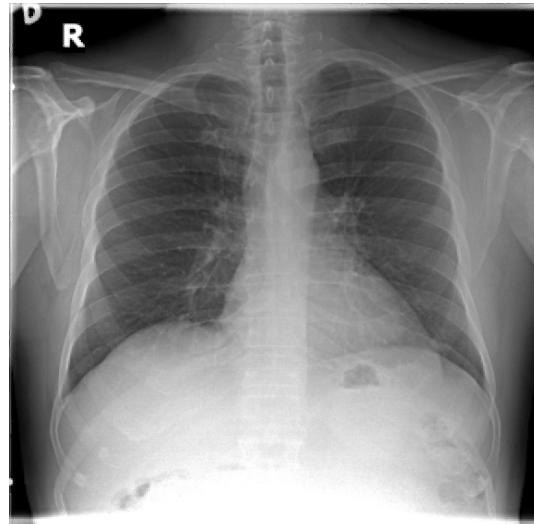


(b) Erreur produite par la carte auto-organisatrice en fonction de  $n$  et  $m$

FIGURE 13 – Résultats obtenus avec la deuxième méthode d’approximation



(a) CAO, 60.5kB,  $n = 7, m = 30$



(b) CAO, 60.5kB,  $n = 7, m = 49$

FIGURE 14 – Images produites par la deuxième méthode d’approximation



FIGURE 15 – CAO, 60.5kB,  $n = 10, m = 100$

## 8.5 Comparaison de temps d'exécution

$m$	Sans approximation en $\mathcal{O}(m^2HL)$	Première méthode d'approximation en $\mathcal{O}(mHL)$	Deuxième méthode d'approximation en $\mathcal{O}(HL \log m)$
2	3 s	< 1 s	< 1 s
10	7 s	1 s	< 1 s
20	24 s	1 s	1 s
30	45 s	3 s	1 s
49	120 s	5 s	1 s
64	206 s	6 s	2 s

FIGURE 16 – Comparaison d'algorithmes pour une image de dimensions  $2000 \times 1976$

## Références

- [1] N. Ahmed et al. Discrete cosine transform. *IEEE Transactions on Computers*, C-23(1) :90–93, 01 1974.
- [2] T. Kohonen. *Self-Organization and Associative Memory*, volume 8. Springer Berlin Heidelberg, 1989.
- [3] M. Rabbani and P. W. Jones. *Digital image compression techniques*. Tutorial texts in optical engineering. Spie Optical Engineering Press, Washington, 1991.

# Quatrième partie

## Annexes

Listing 1 – Implémentation de l'encodeur JPEG pour les images en noir et blanc

```
1 const unsigned char enTete[] = {0xFF,0xD8,0xFF,0xE0,0,0x10, 'J', 'F', 'I', 'F', 0,1,1,0,0,1,0,
2 1,0,0,0xFF,0xDB,0,0x43,0};  
3  
4 // Table de quantification de luminance  
5 const int TQL[] = {16,11,10,16,24,40,51,61,12,12,14,19,26,58,60,55,14,13,16,  
6 24,40,57,69,56,14,17,22,29,51,87,80,62,18,22,37,56,68,109,  
7 103,77,24,35,55,64,81,104,113,92,49,64,78,87,103,121,120,  
8 101,72,92,95,98,112,100,103,99};  
9  
10 // Indice dans l'ordre zigzag de chaque element  
11 const int zigzag[] = {0,1,5,6,14,15,27,28,2,4,7,13,16,26,29,42,3,8,12,17,25,30,  
12 41,43,9,11,18,24,31,40,44,53,10,19,23,32,39,45,52,54,20,22,  
13 33,38,46,51,55,60,21,34,37,47,50,56,59,61,35,36,48,49,57,  
14 58,62,63};  
15  
16 // DC  
17 const unsigned char nbCodesLumDC[16] = {0,1,5,1,1,1,1,1,1,0,0,0,0,0,0,0};  
18 // Nombres de symboles de longueur i (1...,16)  
19 // S = 12  
20 const unsigned char valCodesLumDC[12] = {0,1,2,3,4,5,6,7,8,9,10,11}; // 12 codes  
21  
22 // AC  
23 const unsigned char nbCodesLumAC[16] = {0,2,1,3,3,2,4,3,5,5,4,4,0,0,1,125}; // S = 162  
24 const unsigned char valCodesLumAC[162] = {  
25 0x01,0x02,0x03,0x00,0x04,0x11,0x05,0x12,0x21,0x31,0x41,  
26 0x06,0x13,0x51,0x61,0x07,0x22,0x71,0x14,0x32,0x81,0x91,  
27 0xa1,0x08,0x23,0x42,0xb1,0xc1,0x15,0x52,0xd1,0xf0,0x24,  
28 0x33,0x62,0x72,0x82,0x09,0xa0,0x16,0x17,0x18,0x19,0x1a,  
29 0x25,0x26,0x27,0x28,0x29,0x2a,0x34,0x35,0x36,0x37,0x38,  
30 0x39,0x3a,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x53,  
31 0x54,0x55,0x56,0x57,0x58,0x59,0x5a,0x63,0x64,0x65,0x66,  
32 0x67,0x68,0x69,0x6a,0x73,0x74,0x75,0x76,0x77,0x78,0x79,  
33 0x7a,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8a,0x92,0x93,  
34 0x94,0x95,0x96,0x97,0x98,0x99,0x9a,0xa2,0xa3,0xa4,0xa5,  
35 0xa6,0xa7,0xa8,0xa9,0xaa,0xb2,0xb3,0xb4,0xb5,0xb6,0xb7,  
36 0xb8,0xb9,0xba,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,  
37 0xca,0xd2,0xd3,0xd4,0xd5,0xd6,0xd7,0xd8,0xd9,0xda,0xe1,  
38 0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xf1,0xf2,  
39 0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa  
40 };  
41  
42 const unsigned char enTete2[] = {0xFF,0xDA,0,0x8,1,1,0,0,0x3F,0};  
43  
44 // Tables Huffman  
45 const unsigned short YDCH[256][2] = {{0,2},{2,3},{3,3},{4,3},{5,3},{6,3},{14,4},{30,5},{62,6},  
46 {126,7},{254,8},{510,9}};  
47 const unsigned short YACH[256][2] = {  
48 {10,4},{0,2},{1,2},{4,3},{11,4},{26,5},{120,7},{248,8},  
49 {1014,10},{65410,16},{65411,16},{0,0},{0,0},{0,0},{0,0},  
50 {0,0},{0,0},{12,4},{27,5},{121,7},{502,9},{2038,11},  
51 {65412,16},{65413,16},{65414,16},{65415,16},{65416,16}  
52 ,{0,0},{0,0},{0,0},{0,0},{0,0},{28,5},{249,8},  
53 {1015,10},{4084,12},{65417,16},{65418,16},{65419,16},  
54 {65420,16},{65421,16},{65422,16},{0,0},{0,0},{0,0},  
55 {0,0},{0,0},{0,0},{58,6},{503,9},{4085,12},{65423,16},  
56 {65424,16},{65425,16},{65426,16},{65427,16},{65428,16},  
57 {65429,16},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},{59,6},  
58 {1016,10},{65430,16},{65431,16},{65432,16},{65433,16},  
59 {65434,16},{65435,16},{65436,16},{65437,16},{0,0},{0,0},  
60 {0,0},{0,0},{0,0},{0,0},{122,7},{2039,11},{65438,16},  
61 {65439,16},{65440,16},{65441,16},{65442,16},{65443,16},  
62 {65444,16},{65445,16},{0,0},{0,0},{0,0},{0,0},{0,0},  
63 {0,0},{123,7},{4086,12},{65446,16},{65447,16},{65448,16},  
64 {65449,16},{65450,16},{65451,16},{65452,16},{65453,16},
```

```

65 {0,0},{0,0},{0,0},{0,0},{0,0},{250,8},{4087,12},
66 {65454,16},{65455,16},{65456,16},{65457,16},{65458,16},
67 {65459,16},{65460,16},{65461,16},{0,0},{0,0},{0,0},{0,0},
68 {0,0},{0,0},{504,9},{32704,15},{65462,16},{65463,16},
69 {65464,16},{65465,16},{65466,16},{65467,16},{65468,16},
70 {65469,16},{0,0},{0,0},{0,0},{0,0},{0,0},{505,9},
71 {65470,16},{65471,16},{65472,16},{65473,16},{65474,16},
72 {65475,16},{65476,16},{65477,16},{65478,16},{0,0},{0,0},
73 {0,0},{0,0},{0,0},{506,9},{65479,16},{65480,16},{65481,16},
74 {65482,16},{65483,16},{65484,16},{65485,16},{65486,16},
75 {65487,16},{0,0},{0,0},{0,0},{0,0},{0,0},{1017,10},
76 {65488,16},{65489,16},{65490,16},{65491,16},{65492,16},
77 {65493,16},{65494,16},{65495,16},{65496,16},{0,0},{0,0},
78 {0,0},{0,0},{0,0},{1018,10},{65497,16},{65498,16},
79 {65499,16},{65500,16},{65501,16},{65502,16},{65503,16},
80 {65504,16},{65505,16},{0,0},{0,0},{0,0},{0,0},{0,0},
81 {2040,11},{65506,16},{65507,16},{65508,16},{65509,16},{65510,16},
82 {65511,16},{65512,16},{65513,16},{65514,16},{0,0},{0,0},
83 {0,0},{0,0},{0,0},{0,0},{65515,16},{65516,16},{65517,16},
84 {65518,16},{65519,16},{65520,16},{65521,16},{65522,16},
85 {65523,16},{65524,16},{0,0},{0,0},{0,0},{0,0},{2041,11},
86 {65525,16},{65526,16},{65527,16},{65528,16},{65529,16},
87 {65530,16},{65531,16},{65532,16},{65533,16},{65534,16},
88 {0,0},{0,0},{0,0},{0,0},{0,0}
89 };
90
91
92 const float aASF[] = { 1.0f * 2.828427125f, 1.387039845f * 2.828427125f,
93 1.306562965f * 2.828427125f, 1.175875602f * 2.828427125f,
94 1.0f * 2.828427125f, 0.785694958f * 2.828427125f,
95 0.541196100f * 2.828427125f, 0.275899379f * 2.828427125f };
96
97
98
99 #include <iostream>
100 #include <fstream>
101 #include <cmath>
102 #include <complex>
103 #include "Consts.h"
104 #define cimg_use_png
105 #include "CImg.h"
106 using namespace std;
107 using namespace cimg_library;
108
109 int qualite;
110 unsigned char TL[64], details[18];
111 int hauteur = 1, largeur = 1;
112 float fdtbl[64];
113 int bits, bitCnt;
114 ofstream fout;
115 CImg<float> input;
116 const double PI = acos(-1);
117 float col;
118 bool RGB;
119
120 float Lum(int y, int x) {
121     if (!RGB) return input(x,y,0,0)/32768/2*256-128;
122     float R = input(x,y,0,0);
123     float G = input(x,y,0,1);
124     float B = input(x,y,0,2);
125     return float(0.299 * R + 0.587 * G + 0.114 * B - 128);
126 }
127
128 void imprimerBits(const unsigned short *bs);
129
130 void calcBits(int val, unsigned short bits[2]);
131
132 void precalc() {
133     // Table de quantification
134     for (int i = 0; i < 64; i++) {
135         int valeur = (TQL[i] * qualite + 50) / 100;

```

```

136         valeur = max(valeur, 1);
137         valeur = min(valeur, 255);
138         TL[zigzag[i]] = valeur;
139     }
140
141     // 2eme En-tete
142     unsigned char details1[] = {0xFF,0xC0, // Baseline DCT (Huffman)
143     0,0x0B,
144     8, //nbre de bits codant chaque coefficient
145     (unsigned char)(hauteur>>8),(unsigned char)(hauteur&0xFF),
146     (unsigned char)(largeur>>8),(unsigned char)(largeur&0xFF),
147     1 /*grayscale*/,1,0x11,0, // ic = 1, iq = 0
148     0xFF,0xC4,0,0xD2,
149     0 // DC, iq=0
150   };
151   for (int i = 0; i < 18; i++)
152     details[i] = details1[i];
153
154
155   for(int row = 0, k = 0; row < 8; ++row)
156     for(int col = 0; col < 8; ++col, ++k)
157       fdtbl[k] = 1 / (TL [zigzag[k]] * aASF[row] * aASF[col]);
158 }
159
160 float C(int z) {
161   return (z ? 1 : (float)0.70710678118);
162 }
163
164 void DCTII(float *block) {
165   float dct[64];
166   for (int i = 0; i < 8; i++)
167     for (int j = 0; j < 8; j++) {
168       int p = i*8+j;
169       dct[p] = 0;
170       for (int y = 0; y < 8; y++)
171         for (int x = 0; x < 8; x++) {
172           int pp = y*8+x;
173           dct[p] += float(block[pp]*cos((2*x+1)*j*PI/16)*cos((2*y+1)*i*PI/16));
174         }
175       dct[p] *= (float)0.25*C(i)*C(j);
176     }
177   for (int i = 0; i < 64; i++)
178     block[i] = dct[i];
179 }
180
181 void DCT(float &d0, float &d1, float &d2, float &d3, float &d4, float &d5, float &d6, float &d7) {
182   float tmp0 = d0 + d7;
183   float tmp7 = d0 - d7;
184   float tmp1 = d1 + d6;
185   float tmp6 = d1 - d6;
186   float tmp2 = d2 + d5;
187   float tmp5 = d2 - d5;
188   float tmp3 = d3 + d4;
189   float tmp4 = d3 - d4;
190
191   // Even part
192   float tmp10 = tmp0 + tmp3; // phase 2
193   float tmp13 = tmp0 - tmp3;
194   float tmp11 = tmp1 + tmp2;
195   float tmp12 = tmp1 - tmp2;
196
197   d0 = tmp10 + tmp11; // phase 3
198   d4 = tmp10 - tmp11;
199
200   float z1 = (tmp12 + tmp13) * 0.707106781f; // c4
201   d2 = tmp13 + z1; // phase 5
202   d6 = tmp13 - z1;
203
204   // Odd part
205   tmp10 = tmp4 + tmp5; // phase 2
206   tmp11 = tmp5 + tmp6;

```

```

207     tmp12 = tmp6 + tmp7;
208
209 // The rotator is modified from fig 4-8 to avoid extra negations.
210 float z5 = (tmp10 - tmp12) * 0.382683433f; // c6
211 float z2 = tmp10 * 0.541196100f + z5; // c2-c6
212 float z4 = tmp12 * 1.306562965f + z5; // c2+c6
213 float z3 = tmp11 * 0.707106781f; // c4
214
215 float z11 = tmp7 + z3; // phase 5
216 float z13 = tmp7 - z3;
217
218 d5 = z13 + z2; // phase 6
219 d3 = z13 - z2;
220 d1 = z11 + z4;
221 d7 = z11 - z4;
222 }
223
224 void DCTAAN(float *dct) {
225 // DCT lignes
226 for (int i = 0; i < 64; i += 8) {
227     DCT(dct[i], dct[i+1], dct[i+2], dct[i+3], dct[i+4], dct[i+5], dct[i+6], dct[i+7]);
228 }
229
230 // DCT colonnes
231 for (int i = 0; i < 8; i++)
232     DCT(dct[i], dct[i+8], dct[i+16], dct[i+24], dct[i+32], dct[i+40], dct[i+48], dct[i+56]);
233 }
234
235 int encoder(float *dct, int DC) {
236 const unsigned short EOB[2] = { YACH[0x00][0], YACH[0x00][1] };
237 const unsigned short M16zeroes[2] = { YACH[0xF0][0], YACH[0xF0][1] };
238
239 DCTAAN(dct);
240 //DCTII(dct);
241
242 //DCT de la DCT
243 /*
244 for(int y = 0, j=0; y < 8; ++y) {
245     for(int x = 0; x < 8; ++x,++j) {
246         int i = y*8+x;
247         dct[i] = -abs(dct[i])*fdtbl[i];
248         dct[i] = (dct[i]/2048*256+128);
249     }
250 }
251 DCTAAN(dct);
252 */
253
254 // Quantification, zigzag
255 int YE[64];
256 for(int y = 0; y < 8; ++y) {
257     for(int x = 0; x < 8; ++x) {
258         int i = y*8+x;
259         float v = dct[i]*fdtbl[i];
260 //         float v = dct[i]/TL[i];
261         YE[zigzag[i]] = (int)(v < 0 ? ceilf(v - 0.5f) : floorf(v + 0.5f));
262     }
263 }
264
265 // DC
266 int diff = YE[0] - DC;
267 if (diff == 0) {
268     imprimerBits(YDCH[0]);
269 } else {
270     unsigned short bits[2];
271     calcBits(diff, bits);
272     imprimerBits(YDCH[bits[1]]);
273     imprimerBits(bits);
274 }
275
276 // AC

```

```

278     int end0pos = 63;
279     for( ; (end0pos>0)&&(YE[end0pos]==0); --end0pos) {
280     }
281     // end0pos = first element in reverse order !=0
282     if(end0pos == 0) {
283         imprimerBits(EOB);
284         return YE[0];
285     }
286     for(int i = 1; i <= end0pos; ++i) {
287         int startpos = i;
288         for ( ; YE[i]==0 && i<=end0pos; ++i) {
289             }
290         int nrzeroes = i-startpos;
291         if ( nrzeroes >= 16 ) {
292             int lng = nrzeroes>>4;
293             for (int nrmarker=1; nrmarker <= lng; ++nrmarker)
294                 imprimerBits(M16zeroes);
295             nrzeroes &= 15;
296         }
297         unsigned short bits[2];
298         calcBits(YE[i], bits);
299         imprimerBits(YACH[(nrzeroes<<4)+bits[1]]);
300         imprimerBits(bits);
301     }
302     if(end0pos != 63)
303         imprimerBits(EOB);
304     return YE[0];
305 }
306
307 int main() {
308     cout << "QUALITE(1-100): ";
309     cin >> qualite;
310     cout << "Y/RGB(0/1): ";
311     cin >> RGB;
312     qualite = qualite < 50 ? 5000 / qualite : 200 - qualite * 2;
313     fout.open("image.jpg", ios::binary);
314     input.load("input.png");
315     hauteur = input.height();
316     largeur = input.width();
317     //hauteur = 30000;
318     //largeur = 35000;
319     cout << hauteur << ' ' << largeur << endl;
320     precalc();
321
322     // EN-TETE
323     for (int i = 0; i < 25; i++)
324         fout << enTete[i];
325
326     // TABLE DE QUANTIFICATION DE LA LUMINANCE
327     for (int i = 0; i < 64; i++)
328         fout << TL[i];
329
330     // TYPE DE DCT, LONGUEUR, LARGEUR, NOMBRE DE COMPOSANTS
331     for (int i = 0; i < 18; i++)
332         fout << details[i];
333
334     // TABLE HUFFMAN DE LA LUMINANCE
335     // DC
336     for (int i = 0; i < 16; i++)
337         fout << nbCodesLumDC[i];
338     for (int i = 0; i < 12; i++)
339         fout << valCodesLumDC[i];
340
341     fout << (unsigned char)0x10; // AC, iq = 0
342
343     // AC
344     for (int i = 0; i < 16; i++)
345         fout << nbCodesLumAC[i];
346     for (int i = 0; i < 162; i++)
347         fout << valCodesLumAC[i];
348

```

```

349     // START OF SCAN
350     for (int i = 0; i < 10; i++)
351         fout << enTete2[i];
352
353     float Y[64];
354     int DC = 0;
355     for (int i = 0; i < hauteur; i += 8)
356         for (int j = 0; j < largeur; j += 8) {
357             // BLOC
358             for (int k = i; k < i+8; k++) {
359                 int kk = min(k, hauteur-1);
360                 for (int l = j; l < j+8; l++) {
361                     int ll = min(l, largeur-1);
362                     Y[(k-i)*8+(l-j)] = Lum(kk,ll);
363                 }
364             }
365             DC = encoder(Y, DC);
366         }
367     unsigned short finirBits[] = {0x7F, 7};
368     imprimerBits(finirBits);
369     fout << (unsigned char)0xFF << (unsigned char)0xD9;
370     system("pause");
371 }
372
373
374
375 void imprimerBits(const unsigned short *bs) {
376     bitCnt += bs[1];
377     bits |= bs[0] << (24 - bitCnt);
378     while (bitCnt >= 8) {
379         unsigned char c = (bits >> 16) & 255;
380         fout << c;
381         if(c == 255)
382             fout << (unsigned char)0;
383         bits <= 8;
384         bitCnt -= 8;
385     }
386 }
387
388 void calcBits(int val, unsigned short bits[2]) {
389     int tmp1 = abs(val);
390     val = val < 0 ? val-1 : val;
391     bits[1] = 1;
392     while(tmp1 >>= 1) {
393         ++bits[1];
394     }
395     bits[0] = val & ((1<<bits[1])-1);
396 }
```

---

Listing 2 – Implémentation de la carte auto-organisatrice

```

1 #ifndef NETWORK_H
2 #define NETWORK_H
3 #include <vector>
4 #include <utility>
5 #include <map>
6 #include <algorithm>
7 #include <cmath>
8 #include <iostream>
9 #include <cstdio>
10 #include <io.h>
11 #include <random>
12 using namespace std;
13
14 struct Node{
15     vector<float> w;
16     Node(int sz, mt19937 *rng) {
17         w = vector<float>(sz);
18         uniform_real_distribution<float> distrib(0,255);
19         for (int i = 0; i < sz; i++)
20             w[i] = distrib(*rng);
```

```

21     }
22 };
23
24 struct network{
25     mt19937 *gen;
26     float mu, beta;
27     int counter;
28     int n, p, idi, idj;
29     vector<float> E;
30     vector<pair<float,pair<int,int>>> distances;
31     vector<vector<Node>> X;
32     network(){}
33     network(int nn, int pp, mt19937 *rng, float mmu, float mbeta, int cnt) { // O(np^2)
34         idi = idj = -1;
35         counter = cnt;
36         E = vector<float>(nn);
37         mu = mmu; beta = mbeta;
38         p = pp; n = nn;
39         gen = rng;
40         for (int i = 0; i < p; i++) {
41             vector<Node> temp;
42             for (int j = 0; j < p; j++) {
43                 Node a = Node(n, rng);
44                 temp.emplace_back(a);
45             }
46             X.emplace_back(temp);
47         }
48     }
49     void set_input(vector<float> &pixels) { // O(n)
50         for (int i = 0; i < n; i++)
51             E[i] = pixels[i];
52     }
53     float dist(int i, int j) { // O(n)
54         float dis = 0;
55         int l = 0;
56         for (float k : X[i][j].w) {
57             dis += (k-E[l])*(k-E[l]);
58             ++l;
59         }
60         return dis;
61     }
62     void distc(int i, int j) { // O(n)
63         float dis = 0;
64         int l = 0;
65         for (float k : X[i][j].w) {
66             dis += (k-E[l])*(k-E[l]);
67             ++l;
68         }
69     }
70     void update_weights(int i, int j) { // O(n)
71         for (int x = 0; x < n; x++)
72             X[i][j].w[x] += mu*(E[x]-X[i][j].w[x]);
73         for (int f = -1; f < 2; f++)
74             for (int g = -1; g < 2; g++) {
75                 if (abs(g)+abs(f) != 1 || max(i+f, j+g) >= p || min(i+f, j+g) < 0) continue;
76                 for (int x = 0; x < n; x++)
77                     X[i+f][j+g].w[x] += beta*(E[x]-X[i+f][j+g].w[x]);
78             }
79     }
80     void train() { // O(np^2)
81         float mini = 1e18;
82         idi = -1; idj = -1;
83         for (int i = 0; i < p; i++) {
84             for (int j = 0; j < p; j++) {
85                 float dis = dist(i,j);
86                 if (mini > dis) {
87                     mini = dis;
88                     idi = i;
89                     idj = j;
90                 }
91             }
92         }

```

```

92     mu = max(mu*exp(-0.00005*counter),0.001);
93     beta = max(beta*exp(-0.00005*counter),0.001);
94     if (counter % 100 == 0)
95         mu = 0.5, beta = 0.5;
96     update_weights(idi, idj);
97     counter++;
98 }
99 void compute_output() { // O(np^2)
100     float mini = 1e18;
101     idi = -1; idj = -1;
102     for (int i = 0; i < p; i++)
103         for (int j = 0; j < p; j++) {
104             float dis = dist(i,j);
105             if (mini > dis) {
106                 mini = dis;
107                 idi = i;
108                 idj = j;
109             }
110         }
111 }
112 void compute_output_approx() { // O(np)
113     float mini = 1e18;
114     idi = -1; idj = -1;
115     int cn = 0;
116     uniform_int_distribution<> distrib(0,p-1);
117     while (1) {
118         ++cn;
119         int i = distrib(*gen);
120         int j = distrib(*gen);
121         float dis = dist(i,j);
122         if (mini > dis) {
123             mini = dis;
124             idi = i;
125             idj = j;
126         }
127         if (mini <= n*n || cn > p*5)
128             break;
129     }
130 }
131 void compute_output_approx_sec() { // O(n)
132     float mini = 1e18;
133     idi = -1; idj = -1;
134     int cn = 0;
135     uniform_int_distribution<> distrib(0,p-1);
136     float dist_E = 0;
137     for (int i = 0; i < n; i++)
138         dist_E += E[i]*E[i];
139     auto element = make_pair(dist_E,make_pair(0,0));
140     int k = lower_bound(distances.begin(),distances.end(),element)-distances.begin();
141     for (int l = max(k-20,0); l < min(p*p,k+20); l++) {
142         ++cn;
143         int i = distances[l].second.first;
144         int j = distances[l].second.second;
145         float dis = dist(i,j);
146         if (mini > dis) {
147             mini = dis;
148             idi = i;
149             idj = j;
150         }
151     }
152 }
153 float err() { // O(n)
154     float ans = 0;
155     for (int i = 0; i < n; i++)
156         ans += sqrt((X[idi][idj].w[i]-E[i])*(X[idi][idj].w[i]-E[i]));
157     return ans;
158 }
159 void parse(string name) {
160     freopen(name.c_str(),"r",stdin);
161     cin >> counter;
162     for (int i = 0; i < p; i++)

```

```

163         for (int j = 0; j < p; j++) {
164             float dis = 0;
165             for (int k = 0; k < n; k++) {
166                 cin >> X[i][j].w[k];
167                 dis += X[i][j].w[k]*X[i][j].w[k];
168             }
169             distances.push_back({dis,{i,j}});
170         }
171         sort(distances.begin(),distances.end());
172         fclose(stdin);
173     }
174     void save(string name) {
175         int stdout_dupfd;
176         FILE *temp_out;
177         stdout_dupfd = _dup(1);
178         temp_out = fopen(name.c_str(), "w");
179         _dup2(_fileno(temp_out), 1);
180         cout << counter << endl;
181         for (int i = 0; i < p; i++) {
182             for (int j = 0; j < p; j++) {
183                 for (int k = 0; k < n; k++)
184                     cout << X[i][j].w[k] << ' ';
185                 cout << endl;
186             }
187             fclose(temp_out);
188             _dup2(stdout_dupfd, 1);
189             _close(stdout_dupfd);
190         }
191     };
192
193
194 #endif // NETWORK_H
195
196 #include <iostream>
197 #include <random>
198 #include <vector>
199 #include <algorithm>
200 #include <ctime>
201 #include <filesystem>
202 #include <ctype.h>
203 #include <stdio.h>
204 #include <fstream>
205 #include <string>
206 #include <sstream>
207 #define cimg_use_png
208 #include "CImg.h"
209 #include "Network.h"
210
211 using namespace std;
212 using namespace cimg_library;
213
214 int myrandom (int i) { return std::rand()%i;}
215
216 inline bool exists_test (const std::string& name) {
217     struct stat buffer;
218     return (stat (name.c_str(), &buffer) == 0);
219 }
220
221 int main() {
222     ios::sync_with_stdio(0);
223     cin.tie(0);
224     string file;
225     cout << "File name: ";
226     cin >> file;
227     if (file == "0")
228         file = "input.png";
229     else file += ".png";
230     int bsz, outsz, d = 1;
231     cout << "Block size: ";
232     cin >> bsz;
233     cout << "Output layer size: ";

```

```

234     cin >> outsz;
235     float mu, beta;
236     cout << "mu: ";
237     cin >> mu;
238     cout << "beta: ";
239     cin >> beta;
240     int iter;
241     cout << "Iterations: ";
242     cin >> iter;
243     cout << "Approximate output? (0/1/2): ";
244     int approximate;
245     cin >> approximate;
246     random_device rd;
247     mt19937 rng(rd());
248     CImg<float> input;
249     input.load(file.c_str());
250     int h = input.height();
251     int w = input.width();
252
253     network net(bsz*bsz,outsz,&rng,mu,beta,0);
254     stringstream stream;
255     stream << fixed << setprecision(2) << mu << ',' << beta;
256     string mubeta = stream.str();
257     string network = to_string(bsz)+"x"+to_string(bsz)+","+to_string(outsz)+","+mubeta;
258
259     if (exists_test(network)) {
260         cout << "Detected saved network" << endl;
261         net.parse(network);
262     }
263
264     vector<vector<float>> pixels;
265     float mx = 0;
266     for (int i = 0; i < h; i++) {
267         vector<float> line;
268         for (int j = 0; j < w; j++) {
269             mx = max(mx, input(j,i,0,0));
270             line.push_back(input(j,i,0,0));
271         }
272         pixels.push_back(line);
273     }
274     if (mx > 255) d = 256;
275     vector<vector<float>> inputs;
276     for (int i = 0; i < h; i += bsz) {
277         if (i+bsz > h) i = h-bsz;
278         vector<float> E;
279         for (int j = 0; j < w; j += bsz) {
280             if (j+bsz > w) j = w-bsz;
281             for (int f = 0; f < bsz; f++)
282                 for (int g = 0; g < bsz; g++)
283                     E.push_back(pixels[i+f][j+g]/d);
284             inputs.push_back(E);
285             E.clear();
286         }
287     }
288     auto svin = inputs;
289     int c=1,tot=iter;
290     time_t start;
291     start = time(NULL);
292     while (iter--) {
293         _Random_shuffle1(inputs.begin(),inputs.end(),myrandom);
294         for (auto E : inputs) {
295             net.set_input(E);
296             net.train();
297         }
298         time_t cur = time(NULL);
299         cout << c++ << '/' << tot << ", time elapsed: " << difftime(cur,start) << 's' << endl;
300         if (iter%7 == 0) {
301             net.save(network);
302             cout << "Network saved." << endl;
303         }
304     }
305     CImg<float> img(w,h);

```

```

305     int ii = 0;
306     float error = 0;
307     start = time(NULL);
308     for (int i = 0; i < h; i += bsz) {
309         if (i+bsz > h) i = h-bsz;
310         for (int j = 0; j < w; j += bsz) {
311             if (j+bsz > w) j = w-bsz;
312             net.set_input(svin[ii++]);
313             if (approximate == 1) net.compute_output_approx();
314             else if (approximate == 2) net.compute_output_approx_sec();
315             else net.compute_output();
316             error += net.err();
317             int idi = net.idi;
318             int idj = net.idj;
319             for (int f = 0; f < bsz; f++)
320                 for (int g = 0; g < bsz; g++)
321                     img(j+g,i+f) = net.X[idi][idj].w[f*bsz+g];
322     }
323 }
324 time_t cur = time(NULL);
325 cout << "time elapsed: " << difftime(cur,start) << 's' << endl;
326 cout << error << endl;
327 if (approximate == 0)
328     img.save(("output_" + to_string(bsz) + "_" + to_string(outsz) + ".png").c_str());
329 else if (approximate == 1)
330     img.save(("output_approx_" + to_string(bsz) + "_" + to_string(outsz) + ".png").c_str());
331 else img.save(("output_approx_a_" + to_string(bsz) + "_" + to_string(outsz) + ".png").c_str());
332 system("pause");
333 return 0;
334 }
```

---