

Mathieu FOURCROY
Étudiant N° 11294239

L3 Informatique
Université Paris 8

LICENCE INFORMATIQUE

L'UTILISATION DES CARTES AUTO-ADAPTATIVES DANS LE TRAITEMENT
NUMÉRIQUE DES COULEURS : L'EXEMPLE DE LA POSTÉRISATION D'IMAGES

FOURCROY Mathieu

Sous la direction de Jean-Jacques MARIAGE

Gestionnaire du cours EDF6RNEA

-

RÉSEAUX NEUROMIMÉTIQUES

Année 2014/2015

1. Présentation du projet.....	1
A. L'idée:.....	1
B. L'implémentation:.....	1
C. Environnement de développement:.....	1
2. La postérisation d'images.....	2
A. Qu'est-ce que la postérisation ?.....	2
B. Un exemple d'algorithme.....	3
C. Un exemple d'algorithme concret.....	4
D. Le processus de postérisation.....	5
3. Les cartes auto-adaptatives.....	7
A. Description.....	7
B. Les composants.....	7
1) Les formes d'entrée.....	7
2) Les vecteurs de pondération.....	8
C. Les algorithmes des cartes auto-adaptatives.....	8
1) Initialisation des vecteurs de pondération.....	9
2) Sélection du neurone vainqueur.....	10
3) Modification du voisinage.....	11
D. Conclusion sur les carte auto-adaptatives.....	13
1) Avantages.....	13
2) Inconvénients.....	13
4. Application des cartes auto-adaptatives à la postérisation d'images.....	15
A. Présentation du programme.....	15
B. Fichiers.....	15
1) main.*	15
a) usage().....	15
b) set_vars_from_args().....	15
c) main().....	16
2) arr.*	16
a) arr_sub().....	16
b) arr_add().....	16
c) arr_abs().....	16
d) arr_sum().....	16
e) arr_min_idx().....	17
f) arr_to_IplImage().....	17
3) util.*	17
a) random_sample().....	17
b) random_uint().....	17
c) get_filename_ext().....	17
4) som.*	17
a) son_radius().....	17
b) som_learning_rate().....	18
c) euclidian().....	18
d) compute_distance().....	18
e) som_neighbourhood().....	18
f) compute_delta().....	18

g) som_train()	18
h) som_posterize()	18
C. Structures de données	19
1) Vecteurs	19
2) IplImage	19
D. Usage et exemples	19
E. Fonctionnement du programme	21
1) Les composants	21
a) Les formes d'entrée	21
b) Les vecteurs de pondération	21
2) Les algorithmes	22
a) Initialisation des vecteurs de pondération	22
b) Sélection aléatoire de la forme d'entrée	22
c) Sélection du neurone vainqueur	23
d) Modification du voisinage	26
i. Détermination du voisinage	26
ii. Apprentissage	29
F. Comparaison avec d'autres programmes	32
1) Comparaison avec GIMP	32
2) Comparaison avec picmonkey	34
G. Documentation technique	41
5. Conclusions	42
6. Sources	44

1. Présentation du projet

Présentation à priori du projet :

A. L'idée:

Je voudrais implémenter un programme qui postériser les images avec lesquelles on le nourrit. La postérisation d'images est un effet bien connu, son nom vient du fait que les vieux posters présentaient souvent cet effet qui peut être reproduit dans la plupart des logiciels de traitement d'images (gimp, adobe photoshop, ...). Postériser une image consiste à réduire son nombre de couleurs. En réduisant le nombre de couleurs, les gradations deviennent moins "fluides" et on voit apparaître des coupures de couleurs très nettes. Pour cette raison la postérisation d'image est d'autant plus visible que l'image contient beaucoup de gradations de couleurs.

Je n'ai rien trouvé concernant cette utilisation précise des réseaux de neurones mais j'ai trouvé des articles concernant le classement d'images avec les SOM et pour cela les algorithmes utilisent les couleurs (canaux RGB) des images. Une fois qu'on dispose du vecteur caractérisant les canaux RGB de chaque pixel d'une image on peut en faire ce qu'on en veux. Certains algorithmes de classement d'images créent les classes d'images (clusters) d'après leurs couleurs (d'autres préfèrent utiliser les formes) et déterminent la classe des images d'après une formule de précision que l'on peut utiliser pour situer les principales zones de couleurs différentes d'une image et ainsi la postériser.

B. L'implémentation:

Pour réaliser ce programme je compte utiliser les concepts suivants :

- Extraire les pixels d'une image. OpenCV peut faire cela et il n'est pas compliqué de récupérer les données des pixels d'une image dans un vecteur.
- Utiliser un algorithme de Self Organized Maps. Cette algorithme devra être capable de reconnaître et de grouper les différentes zones de couleurs d'une image. Une zone de couleur étant une zone contenant une majorité de couleurs "proches" par exemple plusieurs nuances de vert dans le feuillage d'un arbre. Cela va permettre de postériser l'image.
- Reconstituer l'image avec les pixels "postérisés".

C. Environnement de développement:

J'utiliserais le langage C qui permettra une exécution plus rapide qu'avec Python mais cela serait tout à fait faisable en Python avec les bindings OpenCV ou le module PIL, de plus la manipulation de vecteurs et tableaux serait beaucoup plus simple en Python mais étant donné le nombre de pixels que le programme pourrait être amené à traiter et étant donné le nombre d'itérations et les opérations effectuées par l'algorithme, le temps d'exécution sera inévitablement beaucoup plus élevé qu'en C. Enfin je ne vois pas spécialement le besoin d'utiliser un langage orienté objet pour ce programme.

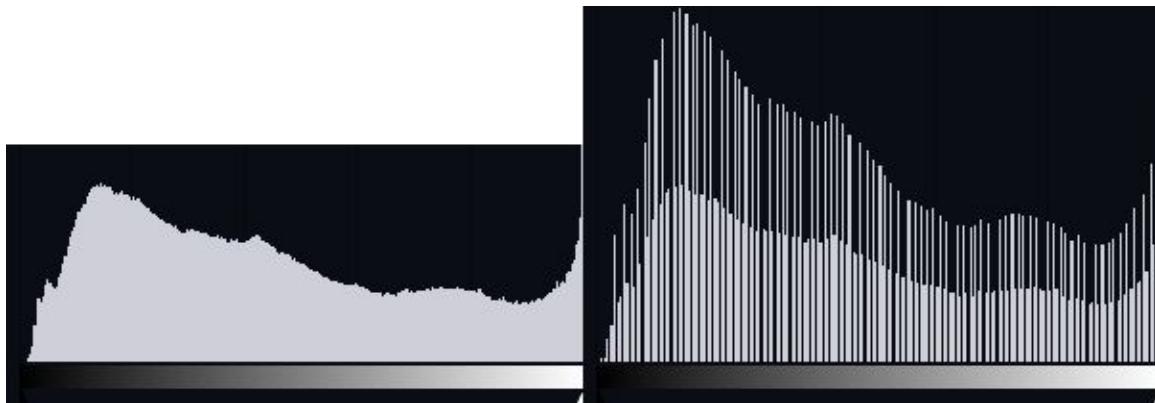
2. La postérisation d'images

A. Qu'est-ce que la postérisation ?

La postérisation se produit lorsque la profondeur de couleurs d'une image a été diminuée, si bien que cela a un impact visuel.

La profondeur des couleurs, dont l'unité est le bit par pixel (bpp), est un terme utilisé pour décrire le nombre de bits utilisés pour représenter la couleur d'un pixel dans une image. Une plus grande profondeur de couleurs, nécessitant un plus grand nombre de bits, permet une plus grande échelle de nuances dans les couleurs. La profondeur (en bpp) quantifie le nombre de couleurs uniques disponibles dans la palette de couleurs d'une image en terme de nombre de 0 et de 1, ou de "bits", qui sont utilisés pour spécifier chaque couleur. Cela ne signifie pas que l'image utilise nécessairement toutes ces couleurs mais qu'elle peut posséder des couleurs avec ce niveau de précision. Pour une image en niveaux de gris, la profondeur de couleurs quantifie le nombre de teintes uniques disponibles. Les images avec des profondeurs de bits plus élevées peuvent coder plus de nuances ou de couleurs car il y a plus de combinaisons de 0 et de 1 disponibles.

Le terme postérisation est utilisé car le procédé permet de modifier l'image de façon à ce qu'elle ressemble à une impression de masse de posters où le processus d'impression utilise un nombre limité d'encre de couleur différentes. Cet effet va du plus subtil au plus prononcé. Tout processus qui "étend" l'histogramme peut créer une postérisation. Ce processus peut lui-même être causé par des techniques logicielles telles que la modification des niveaux et des courbes de couleurs dans Gimp ou Photoshop, ou en convertissant une image d'un espace de couleur dans un autre dans le cadre de la gestion des couleurs. La meilleure façon d'éviter la postérisation non désirée est d'éviter toute manipulation de l'histogramme. Il est possible de repérer la postérisation d'une image à l'œil, cependant le meilleur outil logiciel est l'histogramme. Bien que les histogrammes RVB peuvent montrer des différences extrêmes, les histogrammes de couleurs individuels sont beaucoup plus fiables pour analyser les couleurs d'une image. Les deux histogrammes RVB ci-dessous montrent un cas extrême où un histogramme original a été étendu.



A gauche l'histogramme original, à droite l'histogramme étendu.

Notez le signe révélateur de postérisation sur la droite : les pointes verticales qui

ressemblent aux dents d'un peigne. D'où est-ce que cela provient ? Rappelons que chaque canal dans une image 8-bit ne peut avoir des intensités de couleurs discrètes que dans une fourchette de 0 à 255. (2⁸) Un histogramme étiré est contraint de répartir ces niveaux discrets sur une gamme plus large que celle qui existe dans l'image originale. Cette situation crée des lacunes où il n'y a plus aucune information d'intensité dans l'image. Par exemple, si nous prenons un histogramme de couleur qui varie de 120 à 130 puis si on l'étire de 100 à 150 (5x sa largeur d'origine), alors il y aurait des pics à chaque incrément de 5 (100, 105, 110, etc) et pas de pixels entre les deux. Visuellement, cela forcerait les couleurs à "sauter" ou former des "étapes" là où il y avait précédemment des gradations de couleurs linéaires. Il faut garder à l'esprit que toutes les images numériques ont des niveaux de couleurs discrets (en fonction de leur profondeur de couleurs) et ce n'est que lorsque ces niveaux se dispersent suffisamment que notre œil est capable de les percevoir, comme dans l'image de droite ci-dessous qui correspond à l'image de gauche postérisée.



A gauche l'image originale, à droite l'image postérisée.

La postérisation se produit plus facilement dans les régions de transitions de couleurs progressives, comme un ciel de jour. Ces régions ont besoin de plusieurs niveaux de couleurs pour les représenter et de fait toute diminution des niveaux peut avoir un impact visuel sur l'image. Voyez l'image ci-dessous :



L'image du haut représente une gradation de couleurs linéaire mais celle du bas a été postérisée, le nombre de couleurs a été réduit et de fait on parvient à distinguer nettement les différents tons de couleurs.

B. Un exemple d'algorithme

L'algorithme suivant est un algorithme que j'ai écrit (en pseudo-code) d'après l'algorithme de GIMP (voir section suivante) et d'après un exemple plus concret que j'ai réalisé avec OpenGL et permet de postériser une image.

```

fun posterize (thisImage, thisW, thisH, thisValue)

    numOfAreas = 256.0000 / thisValue
    numOfValues = 255.0000 / (thisValue - 1)

    thisW = thisW - 1
    thisH = thisH - 1

    repeat for y from 0 to thisH
        repeat for x from 0 to thisW

            currentcolor = thisImage.getPixel(x, y)
            currentRed = currentcolor.red
            currentGreen = currentcolor.green
            currentBlue = currentcolor.blue

            redAreaFloat = currentRed / numOfAreas
            redArea = integer(redAreaFloat)
            if redArea > redAreaFloat then
                redArea = redArea - 1
            newRedFloat = numOfValues * redArea
            newRed = integer(newRedFloat)
            if newRed > newRedFloat then
                newRed = newRed - 1

            greenAreaFloat = currentGreen / numOfAreas
            greenArea = integer(greenAreaFloat)
            if greenArea > greenAreaFloat then
                greenArea = greenArea - 1
            newGreenFloat = numOfValues * greenArea
            newGreen = integer(newGreenFloat)
            if newGreen > newGreenFloat then
                newGreen = newGreen - 1

            blueAreaFloat = currentBlue / numOfAreas
            blueArea = integer(blueAreaFloat)
            if blueArea > blueAreaFloat then
                blueArea = blueArea - 1
            newBlueFloat = numOfValues * blueArea
            newBlue = integer(newBlueFloat)
            if newBlue > newBlueFloat then
                newBlue = newBlue - 1

            thisImage.setPixel(x, y, rgb(newRed, newGreen, newBlue))
        end repeat
    end repeat

end fun

```

C. Un exemple d'algorithme concret

Gimp est un logiciel de retouche et modification d'images open source très connu des utilisateurs de Linux. Comme c'est un logiciel open source j'ai décidé de regarder comment il postérisé les images. Voici un extrait du fichier gimpoperationposterize.c :

```

static gboolean
gimp_operation_posterize_process (GeglOperation *operation,
                                void *in_buf,

```

```

        *out_buf,
        glong      samples,
        const GeglRectangle *roi)
{
    GimpOperationPointFilter *point = GIMP_OPERATION_POINT_FILTER (operation);
    GimpPosterizeConfig     *config = GIMP_POSTERIZE_CONFIG (point->config);
    gfloat                  *src   = in_buf;
    gfloat                  *dest   = out_buf;
    gfloat                  levels;

    if (! config)
        return FALSE;

    levels = config->levels - 1.0;

    while (samples--)
    {
        dest[RED_PIX] = RINT (src[RED_PIX] * levels) / levels;
        dest[GREEN_PIX] = RINT (src[GREEN_PIX] * levels) / levels;
        dest[BLUE_PIX] = RINT (src[BLUE_PIX] * levels) / levels;
        dest[ALPHA_PIX] = src[ALPHA_PIX];

        src += 4;
        dest += 4;
    }

    return TRUE;
}

```

Le code est très simple. Chaque pixel est stocké en tant que quadruplet de valeurs (rouge, vert, bleu et alpha) représentées chacune comme un *float* entre 0 et 1. Le canal alpha est simplement copié, tel quel. Les autres canaux sont multipliés par le nombre de niveaux désirés, converties en un nombre entier en arrondissant, multipliés par le nombre de niveaux et stockés dans le canal de pixels de sortie correspondant.

D. Le processus de postérisation

D'après la description et l'exemple d'algorithme ci-dessus on peut définir la postérisation d'image comme un processus qui consiste, très simplement, à réduire le nombre de couleurs d'une image. Il y a des algorithmes très courts et rapides qui permettent de le faire comme celui ci-dessus utilisé par GIMP.

Je vois trois techniques pour postériser une image :

- La première méthode fonctionne sur chacun des trois canaux de couleurs (rouge, vert et bleu) séparément. On divise la gamme de couleurs au sein de chaque canal en bandes égales, puis on adapte chaque pixel de l'image à la bande de couleur le plus proche. Dans cette méthode, on spécifie le nombre de couleurs qu'on veux dans les résultats de la postérisation. L'algorithme offrirait un choix parmi une liste : 8, 27, Si, par exemple, nous voulons diviser chaque canal de couleurs en deux bandes, le nombre total de couleurs possibles dans les trois canaux sera de 8 ($2 \times 2 \times 2$) . De la même manière, 3 bandes par canal créeront 27 couleurs ($3 \times 3 \times 3$), etc... Cette méthode est la plus instinctive et sûrement la plus rapide. Elle est utile pour postériser des images avec des couleurs très distinctes. Mais je pense qu'elle serait à déconseiller pour les photos numériques parce que les couleurs de l'image postérisée pourront être très différentes de celles de l'image originale.

- Une seconde méthode plus recherchée pourrait procéder différemment, en déterminant les couleurs dominantes de l'image. Tout pixel est converti en l'une de ces couleurs dominantes, le plus proche de sa couleur d'origine. Dans cette méthode, nous spécifions le nombre de couleurs que nous voulons dans le résultat, l'image postérisée. Cette méthode est sûrement un peu plus lente, mais les résultats seront meilleurs qu'avec la méthode précédente et surtout plus pertinents dans la plupart des cas. Cette méthode serait appropriée à toutes sortes de photos puisqu'elle affecte les nouvelles couleurs des pixels de sorte à ce qu'elles soient aux plus proche de leur couleur d'origine. La limitation de cette méthode est que si nous avons une image avec de nombreuses couleurs de fond, comme de nombreuses nuances de bleu dans un ciel ou dans un océan, l'algorithme peut favoriser ces couleurs à celles des objets au premier plan et ainsi postériser certains pixels de l'image de telle sorte que leurs couleurs ne correspondent pas du tout à leur couleur d'origine. En définitive cette méthode ne serait pas très adaptée aux images comportant beaucoup de couleurs d'arrière-plan et/ou ayant un arrière-plan occupant beaucoup plus de surface que le premier plan.

- Dans cette troisième méthode il s'agirait de postériser l'image manuellement. Non seulement nous décidons combien de couleurs nous voulons, mais nous choisissons aussi ces couleurs dans l'image. Cette méthode présente l'avantage de laisser le choix des couleurs à l'utilisateur mais a aussi tous les désavantages qui vont avec : cela nécessite des étapes manuelles très lentes et ne permet pas un calcul précis et objectif des couleurs dominantes.

J'ai étudié, à l'occasion de ce projet, les méthodes de postérisation et j'ai expérimenté en écrivant des petits scripts afin d'avoir une idée du procédé, des résultats attendus et de la différence entre les différents résultats, selon les algorithmes utilisés. Cependant, et comme c'est le but de ce cours, le programme que j'ai écrit utilise un tout autre algorithme, un réseau de neurones en fait, qui n'est pas utilisé particulièrement pour la postérisation d'image mais peut servir dans de nombreux problèmes. En réalité je ne connais pas de logiciel utilisant des réseaux de neurones pour postériser des images et c'est aussi ce qui m'a motivé à écrire ce programme. En revanche j'ai pu lire quelques articles sur l'utilisation de réseaux de neurones pour les tâches de reconnaissance de contours dans les images qui réduisent également la profondeur de couleurs des images avant de les "process" [3].

3. Les cartes auto-adaptatives

A. Description

Les cartes auto-adaptatives (Self Organized Maps, SOM) sont une technique de visualisation de données inventée par le professeur Teuvo Kohonen qui réduisent les dimensions de données grâce à l'utilisation de réseaux de neurones auto-adaptatifs. Le problème que la visualisation de données tente de résoudre vient du fait que les hommes ne peuvent tout simplement pas visualiser les données appartenant à des espaces à grandes dimensions. Les SOM réduisent les dimensions en produisant une carte, de généralement une ou deux dimension(s), qui met en évidence les similitudes des données d'entrée en regroupant les éléments similaires. Donc les cartes auto-adaptatives accomplissent deux choses : elles réduisent les dimensions et mettent les similitudes des données en évidence.

Une carte auto-adaptative est donc un type de réseau de neurones qui utilise l'apprentissage non supervisé pour produire avec une faible dimension (généralement une ou deux dimension(s)), la représentation discrétisée de l'espace des données de la forme d'entrée, appelé une carte. Les cartes auto-adaptatives diffèrent des autres réseaux de neurones dans le sens où elles utilisent une fonction de "voisinage" afin de préserver les propriétés topologiques de la forme d'entrée. Comme la plupart des réseaux de neurones, les SOM possèdent deux modes de fonctionnement : l'apprentissage (training) et le test (testing). L'apprentissage construit la carte à l'aide de formes d'entrée, tandis que le test utilise les données construites pour classer automatiquement une nouvelle forme d'entrée inconnue (ou non). Un SOM est constitué de composants appelés nœuds ou neurones. Un vecteur de pondération (weight vector), de la même longueur que les vecteurs de données (ou formes) d'entrée et une position dans l'espace de la carte, sont associé à chaque nœud du réseau. L'agencement habituel de nœuds est un espace à deux dimensions dans une grille rectangulaire. Un SOM représente un espace de données d'entrée à grande dimension en un espace de dimension inférieur.

B. Les composants

1) Les formes d'entrée

La première partie d'un SOM sont les données. L'image ci-dessous représente un exemple de données d'entrée à trois dimensions. Les couleurs sont souvent utilisées dans l'expérimentation avec les SOM. Ici, les couleurs sont représentées en trois dimensions (rouge, vert, et bleu.) L'idée des SOM est de projeter les données à n dimensions (ici les couleurs à 3 dimensions) en quelque chose qui soit visuel et plus simple à comprendre (dans notre exemple, il s'agit d'une image bidimensionnelle). On s'attend donc à ce que les différentes nuances de bleu se retrouvent proches et les orange/jaune devraient également être regroupés ensemble.

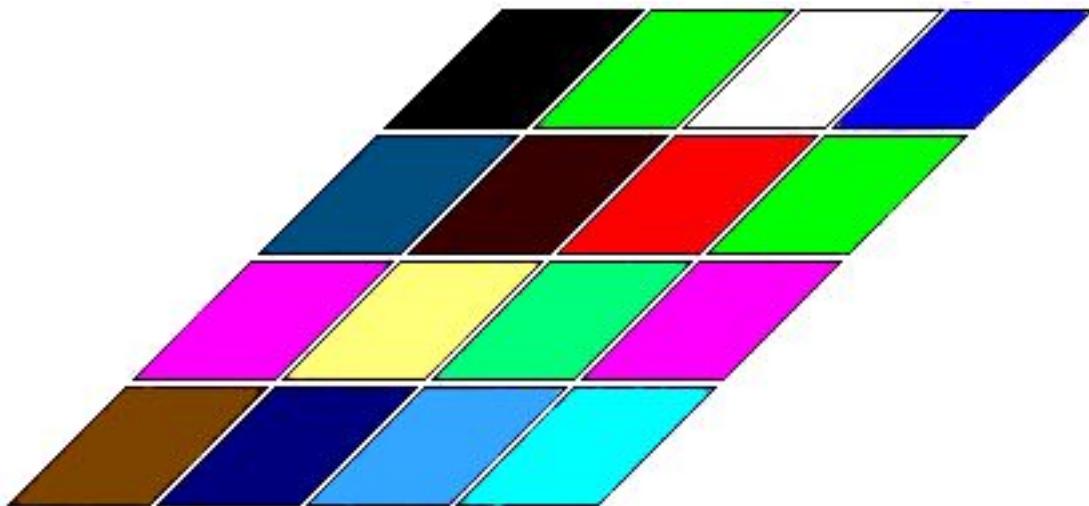


Exemple de formes d'entrée d'un SOM.

Les formes d'entrée sont présentées sous forme de vecteurs à n dimensions. Dans l'exemple des couleurs il s'agit de vecteurs tridimensionnels contenant la valeur des canaux rouge, vert et bleu des pixels.

2) Les vecteurs de pondération

Le second composant d'un SOM sont les vecteurs de pondération. Chaque vecteur de pondération a deux composants propres (cf. Image ci-dessous). La première partie d'un vecteur de pondération sont ses données qui sont de même longueur que les vecteurs d'entrée. La deuxième partie d'un vecteur de pondération est à sa position. Ce qui est pratique avec les couleurs est que les données peuvent être affichées en affichant directement la couleur, de sorte que dans ce cas les couleurs sont les données, et leur emplacement est la position (x, y) du pixel sur l'écran.



Exemple de carte possédant 16 neurones

Dans cet exemple, je essayé de montrer à quoi un tableau en deux dimensions de vecteurs de pondération pourrait ressembler. Cette image est une vue biaisée d'une grille dans laquelle chaque poids (vecteur de pondération) a son propre emplacement unique. La carte des vecteurs de pondération ne doit pas nécessairement être en 2 dimensions, beaucoup d'études ont utilisé des SOM à une seule dimension mais les données des vecteurs de pondérations et les vecteurs d'entrées doivent avoir la même taille.

C. Les algorithmes des cartes auto-adaptatives

Les SOM s'organisent par eux-même grâce à un procédé de compétition pour la représentation des formes d'entrée. Les neurones (les vecteurs de pondération) sont également autorisés à se modifier en apprenant à devenir plus proches des formes d'entrée dans l'espoir de gagner la prochaine compétition. C'est ce processus de sélection et d'apprentissage qui fait que les poids s'organisent en une carte représentant les similitudes des entrées. Donc, avec ces deux composants (les formes d'entrée et les vecteurs de pondération), comment peut-on organiser les vecteurs de pondération de manière à ce qu'ils mettent en évidence les similitudes des vecteurs d'entrée ? Et bien, il faut utiliser l'algorithme ci-dessous.

```

Initialize Map
For t from 0 to 1
    Randomly select a sample
    Get best matching unit
    Scale neighbors
    Increase t by a small amount
End for

```

La première étape dans la construction d'un SOM est d'initialiser les vecteurs de pondération, c'est-à-dire la carte. Puis nous sélectionnons un vecteur d'entrée au hasard et nous recherchons dans la carte des vecteurs de pondération, le vecteur qui représente le mieux le vecteur d'entrée choisi. Comme chaque vecteur de pondération a un emplacement propre, il a aussi des vecteurs de pondération voisins qui sont proches de lui. Le poids (entendez vecteur de pondération) qui est choisi est récompensé en étant capable de devenir plus semblable au vecteur d'entrée, précédemment choisi au hasard. En plus de cette récompense, les voisins de ce poids sont aussi récompensés en étant capable de se rapprocher du vecteur choisi. Après cette étape, nous augmentons t (l'incrément doit être assez petit) parce que le nombre de voisins et la capacité d'apprentissage, de transformation, de chaque poids diminue avec le temps. L'ensemble du processus est répété un grand nombre de fois, 1000 fois minimum dans la plupart des cas.

Dans le cas des couleurs, le programme devra d'abord sélectionner une couleur à partir de l'entrée tel que le vert, puis parcourir les poids de la carte en recherchant l'emplacement contenant la couleur la plus proche de ce vert (je rappelle que les vecteurs de pondération sont initialisés au début de l'algorithme, voir détails dans les sections suivantes). À partir de là, les couleurs voisines au vecteur de pondération ayant remporté la compétition sont rendues "plus vertes", les poids représentant ces couleurs sont modifiés de sorte qu'ils se rapprochent désormais davantage de la forme d'entrée choisie au hasard. Puis une autre couleur est choisie au hasard, comme le jaune, et le processus continue.

1) Initialisation des vecteurs de pondération

Il ya un certain nombre de façon pour initialiser les vecteurs de pondération. Je vois deux façon de procéder:

- La première consiste simplement à donner des valeurs aléatoires aux données de chacun des vecteurs de pondération. Pour illustrer cette méthode, des pixels avec les valeurs de rouge, vert et bleu aléatoires sont représentées ci-dessous.

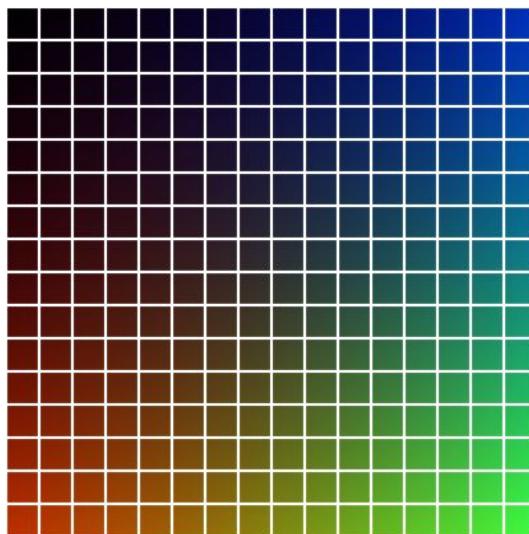
Malheureusement les opérations des SOM sont coûteuses en terme de calculs, je vois donc des variantes de cette méthode d'initialisation des poids qui permettraient d'éloigner

dès le départ les vecteurs dont on sait qu'ils ne sont pas similaires. C'est-à-dire qu'on arrange le voisinage initial de façon à faciliter le travail du réseau. De cette façon, nous aurons besoin de moins d'itérations pour produire le même résultat qu'on obtiendrait avec une initialisation aléatoire.



Exemple d'initialisation aléatoire des vecteurs de pondération.

- Dans cette seconde méthode les valeurs initiales des vecteurs de pondération seront choisies où en tous cas influencées. On peut par exemple, si le but est de grouper des couleurs, donner des valeurs (RGB) entre (0, 0, 0) et (0, 0, 255) aux vecteurs positionnés entre le coin supérieur gauche et le coin supérieur droit, puis faire de même avec des gradations de vert (0, 255, 0) et de rouge (255, 0, 0). L'image suivante illustre mieux ces propos.



Exemple d'initialisation par gradation de couleur des vecteurs de pondération.

2) Sélection du neurone vainqueur

Cette étape est très simple, il suffit de parcourir les vecteurs de pondération et de calculer

la distance entre chaque vecteur et le vecteur d'entrée choisi. Le poids le plus proche de la forme d'entrée et le vainqueur, il remporte la compétition. Si il y a plus d'un vecteur ayant la même distance la plus proche de la forme d'entrée, alors le vainqueur est choisi de façon aléatoire parmi ces vainqueurs potentiels. Il existe plusieurs façons différentes de déterminer la distance entre deux vecteurs. La méthode la plus courante consiste à utiliser la distance euclidienne dont voici la formule :

$$\sqrt{\sum_{i=0}^n x_i^2}$$

Où x_i est la valeur des données du i-ème élément des données d'une forme d'entrée et n est la taille du vecteur.

Dans notre exemple avec les couleurs, si nous considérons les couleurs comme des points 3D alors chaque composant est un axe. Si nous avons choisi le vert (0, 155, 0), le vert clair (70, 220, 70) sera plus proche du vert (0, 155, 0) que le rouge (155, 0, 0).

$$\text{Vert clair} = \sqrt{(70 - 0)^2 + (220 - 155)^2 + (70 - 0)^2} = 14025$$

$$\text{Rouge} = \sqrt{(155 - 0)^2 + (0 - 155)^2 + (0 - 0)^2} = 48050$$

Donc, le vert clair est le vainqueur (Best Matching Unit). Ici nous n'avons testé que deux vecteurs de pondération mais en réalité il faut tous les parcourir. Cette opération est coûteuse (sqrt inside!!).

On peut résumer cette étape dans l'algorithme suivant :

```
fun GetDistance(InputVector, Weights)
    distance = 0
    i = 0
    while i < Weights.size()
        distance += (InputVector[i] - Weights[i]) * (InputVector[i] - Weights[i])
        i++
    return sqrt(distance)
end fun

fun GetBMU(InputVectors, Weights)
    BMUDst = 0
    BMU = None
    for vec in InputVector
        Dst = GetDistance(vec, Weights)
        if Dst > BMUDst
            BMUDst = Dst
            BMU = vec
    return vec
end fun
```

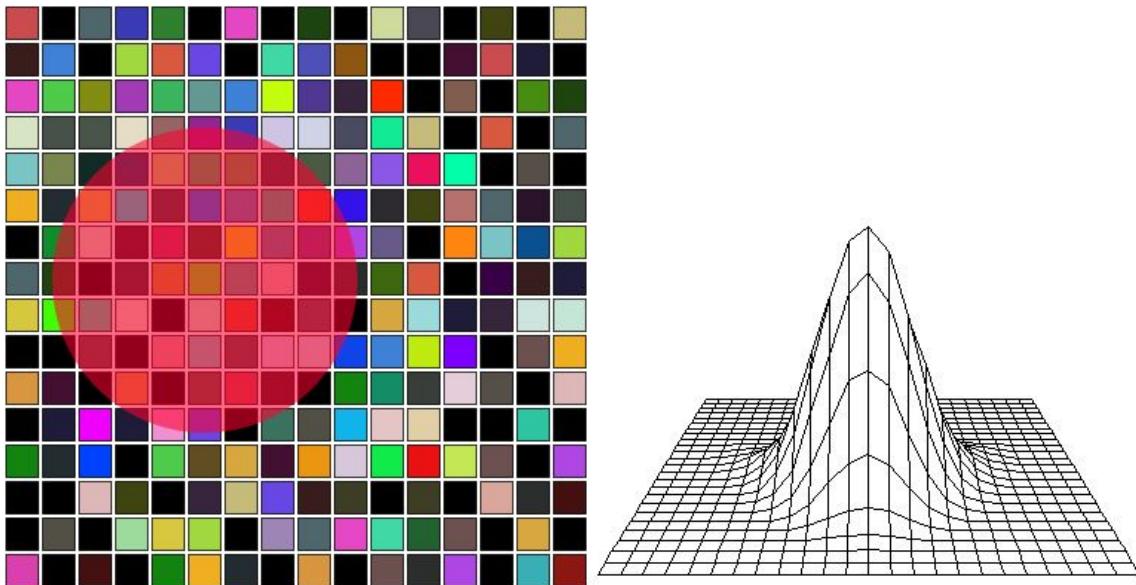
3) Modification du voisinage

a) Détermination du voisinage

Il y a en fait deux parties dans la modification du voisinage : déterminer quels vecteurs de pondération sont considérés comme des voisins et dans quelle proportion chaque vecteur

du voisinage peut se modifier pour se rapprocher du vecteur d'entrée. Les voisins d'un vecteur de pondération vainqueur peuvent être déterminés en utilisant un certain nombre de méthodes différentes. Certaines utilisent des carrés concentriques, d'autres des hexagones, d'autres utilisent une fonction gaussienne où chaque point avec une valeur supérieure à zéro est considéré comme un voisin.

Comme mentionné précédemment, la quantité de voisins diminue au fil du temps. Ceci permet aux formes d'entrée de se placer rapidement dans un sous-espace auquel elles appartiennent très probablement, puis de se positionner plus précisément, on peut voir cela comme un raffinement. Ce processus est similaire à un ajustement grossier suivie d'un réglage plus fin. La fonction utilisée pour diminuer le rayon de voisinage ne compte pas vraiment du moment que ce dernier diminue. Cependant les fonctions linéaires sont préférées.



L'image de gauche est une vue 2D de la carte illustrant le rayon de voisinage centré sur le vecteur correspondant à la couleur vert clair. L'image de droite est une vue 3D de la fonction linéaire utilisée.

L'image de droite montre un graphique de la fonction utilisée. Plus le temps avance, plus la base se rapproche du centre, donc il y a de moins en moins de voisins, au fil du temps. Le rayon initial est souvent une valeur très élevée, proche de la largeur ou de la hauteur de la carte afin de positionner grossièrement mais rapidement les vecteurs d'entrée.

b) Apprentissage

La deuxième partie de l'étape d'ajustement du voisinage est la fonction d'apprentissage proprement dite. Le vecteur de pondération gagnant est récompensé et devient plus proche du vecteur d'entrée choisi. Les voisins deviennent aussi plus proche du vecteur d'entrée choisi. La particularité de ce processus d'apprentissage est que plus le voisin est proche du vainqueur plus il est modifié de façon à ressembler au vecteur d'entrée. En revanche, plus le voisin est éloigné du vainqueur moins il sera modifié afin de ressembler au vecteur d'entrée, mais il sera modifié tout de même, dans ce but. La vitesse à laquelle

un vecteur peut apprendre (sa propension à se modifier) diminue au fil du temps. Utilisons ici une fonction gaussienne. Cette fonction retourne une valeur comprise entre 0 et 1, puis chaque voisin est changé en utilisant l'équation paramétrique. La nouvelle couleur est :

$$\text{Couleur actuelle} * (1.-t) + \text{forme d'entrée} * t$$

Ainsi, dans la première itération, le vainqueur obtiendra $t=1$ pour sa fonction d'apprentissage, de sorte que le vecteur de pondération sera alors exactement le même que la forme d'entrée choisie au hasard. Puisque le rayon de voisinage diminue, la quantité de vecteurs pouvant apprendre s'amoindrie également avec le temps. Sur la première itération, le vainqueur devient identique au vecteur d'entrée choisi car t prend sa valeur dans l'intervalle $[0 ; 1]$ et le vainqueur a donc un degré de similitude égal à 1, c'est à dire qu'il devient une copie du vecteur d'entrée. Mais, alors que le temps passe, la quantité de modification du vainqueur et des voisins diminue, c'est-à-dire qu'ils ne peuvent apprendre autant et donc ils seront modifiés pour ressembler au vecteur d'entrée mais pas autant qu'ils auraient pu l'être au cours d'une itération précédente. La vitesse à laquelle la quantité de modification des vecteurs diminue est linéaire. Pour illustrer cela, dans le graphique précédent, la quantité de données qu'un vecteur peut apprendre est équivalente à la hauteur du "pic" à son emplacement. Au fil du temps, la hauteur de ce "pic" diminue. L'ajout de cette fonction à la fonction de voisinage se traduira par la diminution de la hauteur du "pic" tandis que sa base se rétrécit.

Donc, une fois qu'un vecteur de pondération remporte la compétition, on détermine ses voisins et chacun de ces voisins, en plus du vainqueur va être modifié afin de ressembler davantage au vecteur d'entrée choisi. Plus le vecteur est proche du centre du rayon de voisinage plus il est modifié pour ressembler à la forme d'entrée.

D. Conclusion sur les carte auto-adaptatives

1) Avantages

Les SOM sont très simples à comprendre, comparés à d'autres réseaux de neurones que j'ai pu étudier. Les SOM apportent des résultats souvent pertinents, plus qu'avec d'autres types de réseaux.

2) Inconvénients

Un problème majeur des SOM réside dans l'obtention de données correctes. Nous avons besoin d'une valeur pour chaque dimension de chaque forme d'entrée afin de générer une carte. Parfois, cela est tout simplement impossible et souvent il est très difficile d'acquérir la totalité de ces données. C'est un inconvénient majeur qui limite l'utilisation des SOM et qui est connu sous le nom de "données manquantes" (missing data) [4]. Un autre problème est que chaque SOM est différent et trouve différentes similarités entre les formes d'entrée. Les SOM organisent des formes de telle sorte que dans le résultat, les formes sont généralement entourées par des formes similaires cependant les formes similaires ne sont pas toujours proches les unes des autres. Si nous utilisons beaucoup de nuances de rouge dans les formes d'entrée, nous n'obtiendrons pas toujours un seul grand groupe contenant toutes les nuances de rouge, même après beaucoup d'itérations. Parfois

les groupes seront divisés et il y aura deux groupes de nuances de rouge, voir plus. En utilisant des couleurs comme données, le résultat reste très visuel mais avec n'importe quel autre type de données, il serait très dur de repérer les similitudes entre deux groupes distincts de la carte. Pour cette raison il faut souvent un grand nombre de cartes pour obtenir une très bonne carte.

Mais je pense que le problème majeur des SOM est qu'ils sont très coûteux en calcul, ce qui est un inconvénient important étant donné qu'à mesure que les dimensions des données augmentent, les techniques pour réduire les dimensions deviennent plus importantes, mais, malheureusement, le temps de calcul augmente également. En fait, le nombre de distances que l'algorithme a besoin de calculer augmente de façon exponentielle, ce qui n'en fait pas un algorithme des plus performant...

4. Application des cartes auto-adaptatives à la postérisation d'images

Cette section traite du projet à proprement parlé. J'ai pu le réaliser après avoir étudier les notions exposées dans les sections précédentes. Et comme ces notions ont été, je l'espère, suffisamment explicitées, cette section sera volontairement "courte" puisqu'elle reprend la quasi-totalité de ces notions.

A. Présentation du programme

posternn est un programme écrit en C dans le cadre de ce projet, utilisant la bibliothèque OpenCV et capable de postériser toutes sortes d'images de plusieurs formats différents.

Les formats supportés sont :

- ⌚ jpeg
- ⌚ jpg
- ⌚ jpe
- ⌚ jp2
- ⌚ tiff
- ⌚ tif
- ⌚ png

posternn ne possède aucune interface graphique. Il aurait été très simple d'en ajouter une mais étant donné sa simplicité d'utilisation je pense que le mode ligne de commande est suffisant.

Très simplement, le programme prend en entrée un fichier image, postérisé cette image et renvoie l'image postérissée en sortie.

Les détails de l'implémentation ainsi que les différentes options possibles sont détaillés dans les sections suivantes.

B. Fichiers

1) main.*

Ce fichier contient les fonctions concernant les entrées/sorties c'est à dire les interactions avec l'utilisateur, aussi limitées soit-elles et se charge d'appeler les fonctions de haut niveau nécessaire à la postérisation de l'image.

a) usage()

Se contente d'afficher un message d'usage.

b) set_vars_from_args()

Parse la ligne de commande et détecte les options puis initialise les variables

correspondantes.

c) main()

C'est la fonction principale du programme. Cette fonction appelle set_vars_from_args() puis elle charge l'image spécifiée via l'option -i et en extrait tous ses pixels dans un vecteur à trois dimensions (R, G, B) avec OpenCV (en réalité le vecteur est plat mais utilise des offsets). Ensuite la fonction initialise le SOM, lui donne le vecteur d'entrée : le vecteur contenant les valeurs RGB des pixels afin qu'il apprenne à partir de ce vecteur. Il suffit ensuite d'utiliser la sortie du réseau SOM pour créer la nouvelle image postérisée. La fonction affiche l'image et la sauvegarde.

2) arr.*

J'utilise souvent la bibliothèque CCL (C Containers Library) dans mes programmes C pour profiter des structures de données telles que les vecteurs, les piles, etc... Mais dans ce programme, comme je n'utilise que des vecteurs, j'ai décidé de n'utiliser que la bibliothèque standard. Le code est ainsi un peu moins lisible et plus répétitif mais plus facilement portable. Je n'utilise que les vecteurs comme structure de données mais je les utilise énormément, dans chaque partie du programme. Les réseaux de neurones nécessitent une utilisation intensive des vecteurs... Pour cette raison j'ai rassemblé les fonctions de manipulation de vecteurs dans un fichier séparé.

a) arr_sub()

Soustrait un nombre flottant à chaque éléments d'un vecteur. Il ne s'agit pas d'une soustraction vectorielle entre deux vecteurs mais d'une soustraction d'un flottant propagée dans un vecteur. Par exemple :

```
[4.2, 7.8, 9.0, 1.6] - 2.1 => [2.1, 5.7, 6.9, -0.5]
```

b) arr_add()

Cette fonction calcule la somme de deux vecteurs. Il s'agit d'une addition entre deux vecteurs, une addition vectorielle. Par exemple :

```
[4.2, 7.8, 9.0, 1.6] + [1.7, 6.6, 6.1, 7.0] => [5.9, 14.4, 15.1, 8.6]
```

Note : les deux vecteurs doivent avoir la même taille sinon le résultat n'est pas garanti.

c) arr_abs()

Calcule la valeur absolue d'un vecteur en calculant la valeur absolue de chacun de ses éléments. Par exemple :

```
[4.4, -8.1, -0.5, -1.2] => [4.4, 8.1, -0.5, 1.2]
```

d) arr_sum()

Calcule la somme de tous les éléments d'un vecteur. Par exemple :

```
[4.2, 7.8, 9.0, 1.6] => 22.6
```

e) arr_min_idx()

Recherche l'élément de plus petite valeur dans un vecteur et retourne son index dans ce dernier. Par exemple :

```
[4.2, 7.8, 9.0, 1.6] => 3
```

f) arr_to_IplImage()

Aplatit un vecteur à deux dimensions tout en l'ajoutant à une structure IplImage (définie dans OpenCV). Cette fonction est utilisée dans la fonction main() afin de convertir un vecteur à deux dimensions contenant les valeurs RGB des pixels d'une image (postérisée) en structure IplImage.

Attention cependant, la structure IplImage ne contient pas que les valeurs RGB des pixels, elles contiennent beaucoup d'autres informations. Il faut donc l'initialiser avec les bonnes valeurs auparavant. Cependant, dans le programme, seules les valeurs RGB des pixels ont besoin d'être modifiées, la taille de l'image, son format, etc... ne changent pas. Pour cette raison, plutôt que de copier la structure de l'image originale en remplaçant uniquement les valeurs RGB des pixels je réutilise carrément cette structure puisque de toute façon elle ne sera plus utilisée par la suite.

3) util.*

a) random_sample()

Même si cette fonction est ici utilisée exclusivement dans le but d'initialiser la carte du SOM, elle reste une fonction "utility". Elle se contente de remplir le vecteur qu'on lui passe avec des nombres flottants entre 0 et 1.

b) random_uint()

Retourne un entier (non signé) entre 0 et un maximum donné.

c) get_filename_ext()

Retourne l'extension d'un fichier (par exemple jpg, tif, ...).

4) som.*

Les fonctions suivantes sont directement liées au SOM. Elles implémentent, en C, les algorithmes vu dans la section 3.C..

a) son_radius()

Calcule le rayon de voisinage d'après plusieurs paramètres : le numéro de l'itération actuelle, le nombre maximum d'itérations et la dimension de la carte. Comme expliqué dans la section 3.C.3), le rayon de voisinage diminue à chaque itération. Ici j'utilise une fonction quadratique pour diminuer la valeur du rayon.

b) som_learning_rate()

Calcule la capacité à apprendre des vecteurs. Une fois de plus, ceci a été expliqué dans la section 3.C.3)b). La capacité à apprendre, c'est à dire à se rapprocher de la forme d'entrée, des vecteurs de pondération diminue au fil du temps et des itérations. C'est cette fonction qui se charge de diminuer cette valeur. Dans cette fonction j'utilise une fonction linéaire pour réduire la valeur de la capacité à apprendre des vecteurs.

c) euclidian()

Calcule la distance euclidienne entre chaque vecteur de pondération de la carte et le vecteur d'entrée choisi. Cette fonction est évoquée dans la section 3.C.2), elle sert à déterminer le vecteur de pondération qui remportera la compétition puisque ce sera celui dont la distance est la plus courte, ceci est calculé par arr_min_idx().

d) compute_distance()

Calcule la distance euclidienne entre deux points de l'espace 2D. Cette fonction est mentionnée dans les sections 3.C.2) et 3.C.3). Elle permet, ici, de déterminer le voisinage du vecteur vainqueur (qui est aussi déterminé grâce à sa distance euclidienne par rapport au vecteur d'entrée). Si la distance d'un vecteur de pondération par rapport au vecteur vainqueur est inférieur au rayon de voisinage alors c'est un voisin.

e) som_neighbourhood()

Cette fonction appelle compute_distance() pour chaque vecteur de pondération de la carte afin de déterminer le voisinage du vainqueur (le centre du rayon). Ce processus est détaillé dans la section 3.C.3)a).

f) compute_delta()

Calcule la nouvelle valeur des vecteurs de pondération (les neurones du réseau). Ceux qui font parti du voisinage ainsi que le vainqueur doivent être modifiés afin de se rapprocher davantage du vecteur d'entrée. Le degré de modification dépend de la capacité à apprendre actuelle, calculée par som_learning_rate(). Ce processus est détaillé dans la section 3.C.3)b).

g) som_train()

Cette fonction ne fait qu'appeler des fonctions parmi les précédentes dans une boucle qui itère un certain nombre de fois (c'est le nombre d'étapes d'apprentissage du réseau, couramment appelé epoch). Ainsi, cette fonction réalise l'apprentissage du SOM. Lorsqu'elle a fini elle positionne la carte dans le vecteur tridimensionnel de sortie. Cette fonction correspond à l'algorithme au début de la section 3.C.

h) som_posterize()

Utilise les pixels de l'image originaux ainsi que la sortie du SOM pour créer une table dont les lignes sont les pixels de l'image et les colonnes sont les valeurs de chaque canal

du pixel. La table ainsi créée n'a plus qu'à être aplatie grâce à la fonction `arr_flatten()` pour être dans le même format que le vecteur de pixels de l'image originale extrait avec OpenCV.

C. Structures de données

Le programme n'utilise que deux structures de données. Des vecteurs, beaucoup de vecteurs, pour tout ce qui concerne le SOM. Les réseaux de neurones fonctionnent énormément avec des vecteurs et des matrices. Les données de l'image sont, quant à elles, stockées dans une structure de données spéciale, définie dans OpenCV, `IplImage`.

1) Vecteurs

Quasiment toutes les données du SOM sont stockées dans des vecteurs. Cette structure de données n'existe pas directement en C, il n'y a aucune API standard pour les manipuler. Le programme est un peu parasité par les `malloc()`, les `free()` et surtout par les tests d'allocation de mémoire et je m'en excuse mais tout cela est nécessaire si l'on décide d'utiliser des vecteurs en C.

2) IplImage

Cette structure de données est définie dans la bibliothèque OpenCV. Elle est utilisée lorsqu'on utilise cette bibliothèque pour charger une image en mémoire. Cette structure contient beaucoup d'informations sur l'image chargée. Celles qui nous intéressent et les seules que j'utilise dans le programme sont :

- ⌚ `height` : la hauteur (en pixels) de l'image.
- ⌚ `width` : la largeur (en pixels) de l'image.
- ⌚ `channels` : le nombre de canaux de l'image (le programme ne fonctionne qu'avec les trois canaux RGB).
- ⌚ `imageData` : les données des pixels de l'image. Pour accéder simplement à ces données j'utilise la macro `CV_IMAGE_ELEM` définie par OpenCV. Sinon il est également possible d'y accéder par `offset` mais la macro est là, utilisons la...

D. Usage et exemples

Vous pouvez appeler le programme avec l'option `-h` (`getopt` ne gère pas les options à plus d'un caractère comme `--help` il me semble...) afin d'avoir un message d'usage décrivant très brièvement les différentes options possibles.

```
$ ./posternn -h
```



```

||

This command use a Self Organized Maps algorithm
in order to create a posterization effect on an
image given as input (with -i).
The result is saved as a new image in
<image_name>_posterized.<image_format>
You can specify a custom output file name with -o.
Supported formats are: jpeg, jpg, jpe, jp2, tiff,
tif and png.
But transparency is not correctly handle by default
in OpenCV.

```

```

USAGE: som -i input_file [-l posterization_level]
      [-e number8of8epochs] [-t threshold]
      [-o output_file]
```

```

options description:
  -i Specify the input image to posterize.
  -l Specify the posterization level.
    The higher the level, the more colors.
  -e Specify the number of iterations of the SOM.
  -t Specify the network threshold value.
    If the network delta value ever fall under
    this threshold, the training stop.
  -o Specify the output posterized image path.
```

Seul le chemin vers l'image originale (-i) est obligatoire (évidemment). Si aucun chemin pour l'image postérisée n'est indiqué avec l'option -o alors celle-ci sera enregistrée dans le même répertoire que l'image originale et son nom sera suffixé avec “_posterized”.

Ainsi, vous pouvez appeler le programme de la façon la plus simple avec :

```
posternn -i ./imgs/car.jpg
```

Et l'image postérisée est ./imgs./car_posterized.jpg. Cet exemple fonctionne tel quel puisque le répertoire imgs existe déjà et contient quelques exemples.

Vous pouvez aussi utiliser les différentes options :

- ⌚ -l : Permet d'indiquer le niveau de postérisation à utiliser (2 par défaut). Plus le niveau est élevé plus il y aura de couleur dans l'image et moins l'effet de postérisation sera prononcé. L'argument doit être un nombre entier supérieur à 0. 1 est accepté mais je ne suis pas certain que le résultat sera celui que vous recherchez. En fait la taille de la carte dépend directement de ce paramètre puisque cette valeur définit le nombre de lignes et de colonnes de la carte. Et donc, la carte est obligatoirement carrée (-l 2 = 4 couleurs, -l 3 = 9 couleurs, -l 4 = 16 couleurs, ...).
- ⌚ -e : Permet d'indiquer le nombre d'itérations du SOM. Comme expliqué dans la section 3.C., la valeur minimum, en général, est 1000 itérations. Ici la valeur par défaut est 3000. Vous pouvez paramétrer le SOM pour qu'il effectue n'importe quel nombre d'itérations, du moment que c'est un nombre entier supérieur à 0. Attention cependant, une valeur trop petite ne sera pas suffisante pour que le réseau apprenne

correctement de l'image originale et une valeur trop élevée, en plus d'avoir un temps de calcul assez long, forcera tellement le réseau à modifier ses vecteurs de pondération que les couleurs de l'image postérisée risque d'être assez inappropriées. Comme souvent dans les réseaux de neurones, les réglages sont importants et il faut beaucoup expérimenter avant de trouver de très bons paramètres.

- ☛ -t : Permet d'indiquer le seuil en-dessous duquel la boucle d'apprentissage s'arrêtera. Le seuil est comparé au début de chaque itération avec la somme des valeurs des vecteurs de pondération de la carte.

E. Fonctionnement du programme

Cette section contient des portions du programme, elles sont souvent modifiées afin d'être plus compactes et plus lisibles. Merci de vous référer au code source original.

1) Les composants

La section 3.A) indique que les deux seuls composants d'une carte auto-adaptative sont ses vecteurs d'entrée ainsi que ses vecteurs de pondération. C'est donc sans surprise que cette section reprend ces deux composants.

a) Les formes d'entrée

Une forme d'entrée représente un pixel de l'image. Ainsi le SOM va apprendre d'après chacun des pixels de l'image. Il y a donc autant de formes d'entrée qu'il y a de pixels dans l'image. De plus, chaque vecteur d'entrée possède trois valeurs : la quantité de rouge du pixel, sa quantité de vert et sa quantité de bleu. Ces trois couleurs permettent de décrire n'importe quel couleur opaque (le canal alpha doit être ajouté pour indiquer le niveau d'opacité, on parle de RGBA) mais OpenCV gère mal ce dernier canal par défaut. Les valeurs des canaux RGB prennent, en général, leurs valeurs dans l'intervalle [0, 255] (Cf. Section 2.A.). Ainsi les formes d'entrée ont la forme :

```
Input = [[R0, G0, B0], [R1, G1, B1], [R2, G2, B2], [R3, G3, B3], ..., [Rx, Gx, Bx]]
```

Les valeurs R, G et B de ces vecteurs doivent, pour pouvoir être correctement interprétées par le SOM, être comprises entre 0 et 1 mais OpenCV retourne des valeurs comprises entre 0 et 255. C'est pourquoi elles sont préalablement "clamped", divisées par 255.

b) Les vecteurs de pondération

La seule chose qui puisse paraître déroutante dans ce programme est le fait que les vecteurs de pondérations de la carte ne sont pas stockés sous la forme décrite dans la section 3. C'est-à-dire en tant que vecteurs séparés semblables aux vecteurs d'entrée présentés dans la section précédente. On peut s'attendre à ce que les 16 neurones de taille trois d'une carte de 4 x 4 neurones soient stockés dans 16 vecteurs de taille trois, tel que :

```
Carte = [[R0, G0, B0], [R1, G1, B1], [R2, G2, B2], [R3, G3, B3], ..., [R15, G15, B15]]
```

Mais en fait ce n'est pas le cas. Pour des questions pratiques et surtout pour alléger le code et augmenter ses performances la carte ressemble à (transposition matricielle) :

```
Carte = [[R0, R1, R3, ..., R15], [[G0, G1, G3, ..., G15], [[B0, B1, B3, ..., B15]]
```

Cela n'améliore pas forcément la lisibilité mais étant donné que l'on est très souvent amené à faire des calculs entre chaque canal de couleurs (R, G ou B) ce serait une perte de temps de calcul importante d'avoir à générer de grandes quantités de vecteurs regroupant les valeurs de R, de G et de B.

2) Les algorithmes

L'algorithme principale du réseau SOM est celui de la section 3. :

```
Initialize Map                                     => a)  
For t from 0 to 1  
    Randomly select a sample                   => b)  
    Get best matching unit                   => c)  
    Scale neighbors                         => d)  
    Increase t by a small amount           => e)  
End for
```

Et tout cela est réalisé dans la fonction som_train().

a) Initialisation des vecteurs de pondération

Cette partie du programme correspond à la section 3.C.1).

Les vecteurs de pondération sont initialisés dans la fonction som_train(). Les deux étapes (l'initialisation et l'apprentissage) sont souvent séparées mais j'ai décidé de les regrouper dans la même fonction puisque les vecteurs ne sont initialisés qu'une seule fois et il n'y a qu'un seul apprentissage effectué. De plus les vecteurs de pondération sont forcément initialisés avant l'apprentissage. Voici la partie de la fonction som_train() qui se charge d'initialiser les vecteurs de pondération :

```
/* Randomly initialize weight vectors */  
rand(time(NULL));  
random_sample(WR, nbNeurons);  
random_sample(WG, nbNeurons);  
random_sample(WB, nbNeurons);
```

Et voici à quoi ressemble la fonction random_sample() :

```
void random_sample(float *arr, size_t size){  
    size_t i;  
    for(i = 0; i < size; i++){  
        arr[i] = (float)rand() / (float)(RAND_MAX / 1);  
    }  
}
```

Cette fonction attend un vecteur de float ainsi que sa taille. Elle va remplir ce vecteur avec des flottants entre 0 et 1. Elle est appelée trois fois pour initialiser les trois vecteurs regroupant les valeurs R, G et B de la carte. Je rappelle que la taille de ces vecteurs est égal au nombre de neurones de la carte.

Ainsi l'un des trois vecteurs WR, WG ou WB (W comme weight), s'il est de taille 16 (niveau de postérisation = 4), pourra ressembler à :

```
InitVector = [0.891097, 0.399885, 0.357368, 0.060410, 0.065242, 0.441938, 0.257265,  
0.940351, 0.218373, 0.592225, 0.800584, 0.624896, 0.492458, 0.749472,  
0.126618, 0.644711]
```

b) Sélection aléatoire de la forme d'entrée

À partir d'ici on est dans la boucle d'apprentissage. Il faut, à chaque début de tour, choisir une forme d'entrée aléatoirement parmi tous les vecteurs d'entrée (un vecteur pour chaque pixel de l'image originale). Le bout de code suivant fait cela :

```
/* Randomly choose an input form */
pick = random_uint(nbPixels);
pickRGB[0] = imgPixels[pick][0];
pickRGB[1] = imgPixels[pick][1];
pickRGB[2] = imgPixels[pick][2];
```

La fonction `random_uint()` est très simple :

```
unsigned int random_uint (unsigned int max){
    return (unsigned int)rand() % max;
}
```

Ici on choisit aléatoirement un entier positif entre 0 et `nbPixels`. Sachant que le vecteur à deux dimensions `imgPixels` est de taille `nbPixels` puisqu'il contient les formes d'entrée (une forme pour chaque pixel de l'image originale), le nombre choisi aléatoirement sera forcément un indice valide dans ce vecteur. Si `pick` est l'indice choisi aléatoirement et puisque `imgPixels` est un vecteur 2D alors `imgPixels[pick]` désigne un vecteur. Ce vecteur contient les trois valeurs R, G et B du pixel d'indice `pick`. On remplit le vecteur de taille 3, `pickRGB`, avec les valeurs R, G et B de la forme d'entrée choisie aléatoirement, autrement dit avec les valeurs `imgPixels[pick][0:2]`.

Le vecteur choisi sera toujours de taille 3 (RGB) et pourra, par exemple, être :

```
ChoosedVector = [0.827451, 0.854902, 0.929412]
```

Dans ce cas le pixel correspondant est de couleur bleu très clair : [comme ceci](#).

En effet $0.827451 * 255 = 211$; $0.854902 * 255 = 218$; $0.929412 * 255 = 237$

(R=211, G=218, B=237) = [#D3DAED](#)

c) Sélection du neurone vainqueur

Cette partie du programme correspond à la section 3.C.2).

Nous sommes toujours dans la boucle d'apprentissage. Maintenant que nous avons choisi aléatoirement un vecteur d'entrée correspondant aux valeurs RGB d'un pixel de l'image originale nous mettons tous les vecteurs de pondération de la carte en compétition. Le vecteur dont la distance euclidienne entre lui et le vecteur d'entrée choisi est la plus courte remportera la compétition. Pour déterminer le vainqueur nous devons donc, au préalable, calculer la distance euclidienne entre chaque vecteur de pondération et la forme d'entrée choisie. Voici la partie du programme qui s'occupe de cela, toujours dans la fonction `som_train()` :

```
/* Compute every vectors euclidian distance from the input form */
euclidian(dists, nbNeurons, WR, WG, WB, pickRGB);
```

Voici la fonction `euclidian()` :

```
static int euclidian(float *res, size_t size, float *x, float *y, float *z,
                     float *RGB){
    float *xsub = malloc(sizeof(float) * size);
    float *ysub = malloc(sizeof(float) * size);
    float *zsub = malloc(sizeof(float) * size);
    float *sum = malloc(sizeof(float) * size);
    float r;
```

```

size_t i;

memset(xsub, 0, size);
memset(ysub, 0, size);
memset(zsub, 0, size);

arr_sub(xsub, x, size, RGB[0]);
arr_sub(ysub, y, size, RGB[1]);
arr_sub(zsub, z, size, RGB[2]);

for(i = 0; i < size; i++){
    sum[i] = xsub[i] + ysub[i] + zsub[i];
    float r = (float)sqrt(sum[i]);
    memcpy(&res[i], &r, sizeof(float));
}

free(sum);
free(xsub);
free(ysub);
free(zsub);

return SOM_OK
}

```

Cette fonction attend :

- ⌚ *res*: Un vecteur dans lequel stocker les résultats. Il doit être de la même taille que la carte (nombre de neurones).
- ⌚ *size*: La taille de ce vecteur (nombre de neurones de la carte).
- ⌚ *x*: Les valeurs R de chaque neurone de la carte (d'où l'intérêt de les regrouper par valeurs R, G et B).
- ⌚ *y*: Les valeurs G de chaque neurone de la carte.
- ⌚ *z*: Les valeurs B de chaque neurone de la carte.
- ⌚ *RGB*: Les valeurs R, G et B du vecteur d'entrée choisi précédemment.

Même si la fonction ne ressemble pas à la formule de la distance euclidienne présentée plus haut c'est pourtant bien ce qu'elle calcule. La boucle *for* permet de parcourir tous les vecteurs de la carte et de calculer leur distance euclidienne par rapport au vecteur d'entrée choisi (RGB).

Ainsi le vecteur *res*, en admettant que la carte possède 16 neurones, pourra ressembler à :

```
WeightsDistances = [0.465971, 1.112076, 0.473660, 0.812379, 0.795123, 0.941786, 0.370788,
0.748441, 0.860018, 0.730016, 0.732709, 0.685033, 0.777329, 0.530064,
0.471494, 0.283774]
```

À partir de là il est extrêmement simple de trouver le vainqueur, c'est celui dont la distance est la plus faible. Comme les distances du vecteur calculé ci-dessus y sont ajoutées dans l'ordre, Les indices des vecteurs de valeurs R, G et B des vecteurs de pondération et ceux du vecteur des distances se correspondent. En d'autres mots, dans la fonction ci-dessus, *[x[i], y[i], z[i]]* désignent le i-ème vecteur de pondération et *res[i]* désigne sa distance euclidienne par rapport au vecteur d'entrée choisi. Donc pour trouver le vainqueur il suffit d'identifier la plus petite valeur du vecteur des distances et de retourner son indice. C'est ce qu'on fait dans la portion de code suivante (toujours dans

```
som_train()):
```

```
/* Determine the BMU */  
choosen = arr_min_idx(dists, nbNeurons);  
choosen_x = (int)choosen % mapWidth;  
choosen_y = choosen / mapHeight;
```

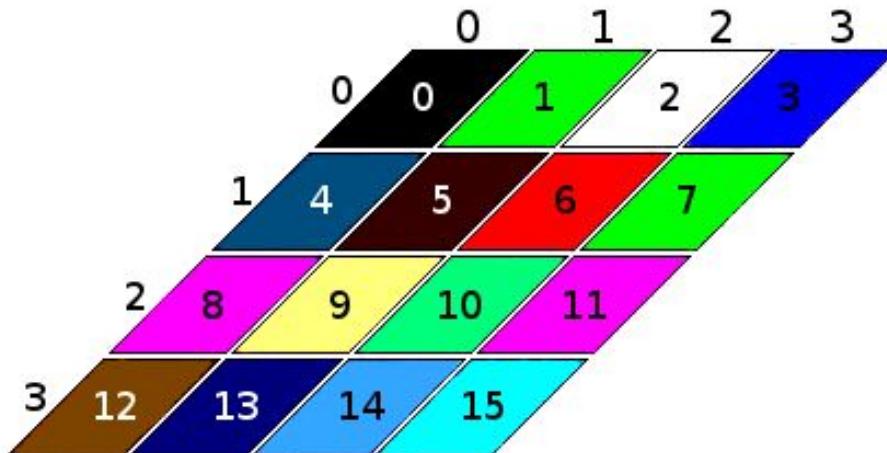
L'indice du vainqueur est retourné par la fonction arr_min_idx() que voici :

```
size_t arr_min_idx(const float *arr, size_t size){  
    size_t i;  
    float minimum = arr[0];  
    size_t idx = 0;  
    for (i = 1; i < size; ++i){  
        if (minimum > arr[i]){  
            minimum = arr[i];  
            idx = i;  
        }  
    }  
    return idx;  
}
```

Elle possède deux paramètres : un vecteur de float (ici le vecteur des distances euclidiennes) et la taille de ce vecteur. Elle mémorise la plus petite valeur du vecteur ainsi que son indice et retourne ce dernier. De cette façon la variable *choosen* contient désormais l'indice du vecteur de pondération le plus proche du vecteur d'entrée choisi, si la carte possède 16 vecteurs de pondération, l'indice du vainqueur est compris entre 0 et 15, par exemple :

```
BMU = 11
```

Pour situer ce vecteur dans la carte on peut reprendre l'illustration de la carte à 16 neurones présentée précédemment. Les couleurs sont des exemples.



On peut voir que le vecteur d'indice 11 dans la carte est dans la colonne 3, ligne 2, on dira plutôt que ces coordonnées 2D sont (3, 2). Et c'est ce qu'on calcule dans le programme (*choosen_x* et *choosen_y*) qui représente donc les coordonnées 2D du vecteur vainqueur dans la carte.. Ainsi on a :

```
BMUX = 3  
BMUY = 2
```

d) Modification du voisinage

Maintenant que nous avons déterminé le vecteur de pondération vainqueur, la prochaine étape, dans la boucle d'apprentissage, est la détermination et la modification du vainqueur et de son voisinage.

i. Détermination du voisinage

Avant de modifier les vecteurs de pondération concernés, il faut d'abord les identifier. Le vecteur vainqueur et ceux de son voisinage doivent être modifiés. Nous connaissons déjà le vecteur vainqueur mais nous ignorons encore son voisinage. Ce dernier dépend du rayon de voisinage (cf. section 3.C.3)a) pour savoir à quoi cela correspond) qui est calculé à la ligne suivante (toujours dans som_train()) :

```
/* Compute the new neighbouring radius */  
rad = som_radius(it, noEpoch, mapWidth, mapHeight);
```

La fonction som_radius() est :

```
static float som_radius(int iterNo, int iterCount, int width, int height){  
    float totalrange = max(width, height) / 2.;  
    float step = iterNo / (float)iterCount;  
    return totalrange - pow(step, 2) * totalrange;  
}
```

Comme expliqué dans la section 3.C.3)a), le rayon de voisinage diminue d'itération en itération. Pour le calculer il faut donc qu'on connaisse le numéro de l'itération actuelle, le nombre total d'itérations à effectuer dans l'algorithme afin de ne pas diminuer le rayon trop rapidement ou trop lentement. La fonction attend également les dimensions de la carte.

La première ligne de la fonction cherche le maximum de la largeur et de la hauteur de la carte. Je vous l'accorde, ceci est inutile puisque dans le programme la carte du réseau est carrée mais ainsi la fonction reste générique. Puis la fonction calcule la "progression" de l'apprentissage en divisant le numéro de l'itération actuelle par le nombre total d'itérations attendues. On obtient ainsi un nombre compris en 0 et 1 qui va servir à la ligne suivante à calculer le rayon de voisinage et à le retourner.

Par exemple, admettons que l'algorithme soit à sa 1856 ième itération et qu'il est sensé en faire 3000 (par défaut). Aussi gardons la taille de la carte des exemples précédents, soit 4 x 4 neurones. Dans ce cas on a :

```
TotalRange = max(4, 4) / 2 = 4 / 2 = 2  
Step = 1856 / 3000 = 0.61866667  
NeighbouringRadius = 2 - 0.61866667^2 * 2 = 1,23450311
```

Ici j'ai décidé d'utiliser une fonction quadratique mais d'autres fonctions sont possibles, bien que celle-ci soit la plus courante.

Nous avons calculé le rayon de voisinage, nous allons maintenant l'utiliser pour découvrir quels sont les voisins du vainqueur. Dans som_train() c'est la ligne juste après

qui s'en charge :

```
/* Find the BMU neighbours */
som_neighbourhood(neigh, choosen_x, choosen_y, rad, mapWidth, mapHeight);
```

La fonction som_neighbourhood() est :

```
static void som_neighbourhood(float *neigh, int x, int y, float radius,
                               int width, int height){
    int i, j, k = 0;
    float distance;
    for(i = 0; i < height ; i++){
        for(j = 0; j < width ; j++){
            distance = compute_distance(i, y, j, x);
            if(distance <= radius){
                neigh[k] = 1 - distance / (float)radius;
            }
            else{
                neigh[k] = 0;
            }
            k++;
        }
    }
}
```

Cette fonction appelle compute_distance() :

```
static float compute_distance(int i, int y, int j, int x){
    return sqrt(pow(i - y, 2) + pow(j - x, 2));
}
```

Je pense pas qu'il soit nécessaire d'expliquer cette dernière fonction. Elle calcule la distance euclidienne entre le point (i, j) et le point (x, y).

som_neighbourhood() attend plusieurs arguments :

- ⌚ *neigh* : un vecteur de la même taille que la carte (nombre de neurones).
- ⌚ *x*: l'abscisse du centre du rayon, c'est-à-dire l'abscisse du vecteur vainqueur.
- ⌚ *y*: l'ordonnée du centre du rayon, c'est-à-dire l'ordonnée du vecteur vainqueur.
- ⌚ *radius* : le rayon de voisinage renvoyé précédemment par som_radius().
- ⌚ *width*: la largeur de la carte (nombre de colonnes).
- ⌚ *height*: la hauteur de la carte (nombre de lignes).

Cette fonction est simple. Comme expliqué dans la section 3.C.3)A), pour chaque vecteur de pondération de la carte on calcule la distance qui le sépare des coordonnées (x, y) du vecteur vainqueur qui est le centre du rayon. Puis, si cette distance est à l'intérieur du rayon de voisinage calculé à l'étape précédente on l'inclue dans le voisinage, sinon on l'ignore.

En réalité, afin que les indices du vecteur de voisinage résultant de cette fonction coïncident avec ceux des autres vecteurs utilisés par la carte (vecteurs des valeurs R, G et B des neurones) et aussi pour ne pas avoir à le ré-allouer à chaque itération, celui-ci est de même taille que la carte (taille 16 si 16 neurones dans la carte). De fait, il agit comme un masque : on y ajoute des décimaux entre 0 (ce vecteur ne fait pas parti du voisinage) et 1 (ce vecteur est le centre du voisinage, c'est le vainqueur) aux indices du vecteur de voisinage. Dit autrement, à chaque neurones i de la carte correspond un degré de voisinage à l'indice i du vecteur de voisinage. Ce degré est 0 si le neurone n'est pas dans le rayon de voisinage ou bien il est supérieur à 0 et plus le neurone est proche du centre du rayon, plus le degré s'approche de 1. Voyez plutôt l'exemple ci-après.

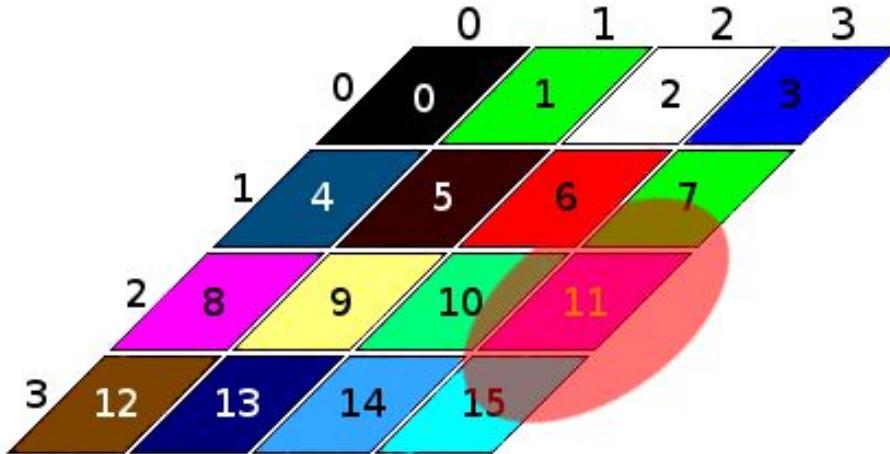
Si nous conservons les données des exemples précédents nous connaissons :

```
x = BMUX = 3
y = BMUY = 2
radius = NeighbooringRadius = 1,23450311
width = 4
height = 4
```

Alors la fonction `som_neighbourhood()` va remplir le tableau `neigh` de tel façon qu'il ressemblera à :

```
Neighboors = [0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000,
              0.189957, 0.000000, 0.000000, 0.189957, 1.000000, 0.000000, 0.000000,
              0.000000, 0.189957]
```

On voit que la valeur à l'indice 11 du vecteur de voisinage est 1, c'est le vainqueur, le centre du rayon. Les valeurs aux indices 7, 10 et 15 ont une valeur supérieur à 0 : ces neurones sont dans le rayon de voisinage. Les autres neurones sont hors du voisinage. On peut voir cela en dessinant le rayon de voisinage sur l'illustration de la carte :



Le cercle (tronqué) rouge représente le rayon de voisinage. On voit qu'il est centré sur le neurone vainqueur (11) et qu'il recouvre de façon assez importante les neurones 7, 10 et 15. Ce sont, par ailleurs, les trois neurones adjacents au neurone 11.

ii. Apprentissage

La modification, l'adaptation, des vecteurs de pondération de la carte est la dernière étape de l'algorithme d'apprentissage et c'est la plus importante.

Comme détaillé dans la section 3.C.3)b), l'apprentissage des neurones dépend de leur capacité à apprendre. Cette capacité à apprendre, à se rapprocher de la forme d'entrée, diminue au fil du temps et des itérations. C'est la fonction som_learning_rate() qui se charge de diminuer cette valeur. Dans cette fonction j'utilise une fonction linéaire pour réduire la valeur de la capacité à apprendre des vecteurs. Voici comment cette fonction est appelée dans som_train():

```
/* Compute the new learning rate */  
eta = som_learning_rate(it, noEpoch);
```

Cette fonction ressemble beaucoup à la fonction som_radius() qui calcul le nouveau rayon de voisinage mais elle est linéaire. On lui passe le numéro de l'itération actuelle et le nombre total d'itérations attendues. Et voici la fonction en question :

```
static float som_learning_rate(int iterNo, int iterCount){  
    float MAX_VALUE = 0.75;  
    float MIN_VALUE = 0.1;  
    float totalrange = MAX_VALUE - MIN_VALUE;  
    float step = iterNo / (float)iterCount;  
    return MAX_VALUE - step * totalrange;  
}
```

À partir de ses deux paramètres la fonction calcule la nouvelle capacité d'apprentissage des neurones. Les bornes maximum et minimum sont des (fausses) constantes. La fonction calcule d'abord la différence entre la borne supérieur et la borne inférieur. Puis elle calcule la "progression" de l'apprentissage en divisant le numéro de l'itération actuelle par le nombre total d'itération attendues, comme dans som_radius(). On obtient ainsi un nombre compris en 0 et 1 qui va servir à la ligne suivante à calculer la nouvelle capacité d'apprentissage qui sera retournée.

Contrairement au cas étudié dans la section 3., ici même à la première itération de la boucle d'apprentissage, le neurone vainqueur ne pourra apprendre 100% des données de la forme d'entrée choisie puisque le maximum est borné par *MAX_VALUE*. Il ne pourra apprendre que 75% de la forme.

Par exemple, admettons que l'algorithme soit à sa 1856 ième itération et qu'il est sensé en faire 3000 (par défaut). Aussi gardons la taille de la carte des exemple précédents, soit 4 x 4 neurones. Dans ce cas on a :

```
TotalRange = 0.75 - 0.1 = 0.65  
Step = 1856 / 3000 = 0.618666667  
LearningRate = 0.75 - 0.618666667 * 0.65 = 0,347867
```

Cela signifie que les neurones pourront devenir 34.79% plus similaire à la forme d'entrée choisie à la première étape de la boucle d'apprentissage.

Ici j'ai décidé d'utiliser une fonction linéaire mais d'autres fonctions sont possibles, bien que celle-ci soit la plus courante.

La suite de la fonction som_train() utilise cette capacité d'apprentissage pour calculer, effectivement, les nouvelles valeurs des neurones de la carte.

```
/* Compute new value of the network weight vectors */
compute_delta(deltaR, eta, neigh, nbNeurons, pickRGB[0], WR);
compute_delta(deltaG, eta, neigh, nbNeurons, pickRGB[1], WG);
compute_delta(deltaB, eta, neigh, nbNeurons, pickRGB[2], WB);
```

La fonction compute_delta() est :

```
void compute_delta(float *res, float eta, float *neigh, size_t nbNeigh,
                  float chan, float *chanArr){
    size_t i;
    for(i = 0; i < nbNeigh; i++){
        res[i] = eta * neigh[i] * (chan - chanArr[i]);
    }
}
```

Elle attend :

- ⌚ res : le vecteur qui contiendra les résultats.
- ⌚ eta : la capacité d'apprentissage que nous venons de calculer.
- ⌚ neigh : le vecteur de voisinage.
- ⌚ nbNeigh : la taille du vecteur de voisinage.
- ⌚ chan : l'une des trois valeurs R, G ou B du vecteur d'entrée choisi au début de la boucle.
- ⌚ chanArr : le vecteur regroupant les valeurs R, G ou B des neurones.

La fonction est un peu compacte mais si on voulait détailler les étapes du calcul on pourrait la remplacer par :

```
void compute_delta(float *res, float eta, float *neigh, size_t nbNeigh,
                  float chan, float *chanArr){
    size_t i;
    float resNeigh[nbNeigh];
    float resChanArr[nbNeigh];

    for(i = 0; i < nbNeigh; i++){
        resNeigh[i] = eta * neigh[i];
    }
    for(i = 0; i < nbNeigh; i++){
        resChanArr[i] = chan - chanArr[i];
    }
    for(i = 0; i < nbNeigh; i++){
        res[i] = resNeigh[i] * resChanArr[i];
    }
}
```

resNeigh contient la multiplication de chaque valeur de voisinage par la valeur de la capacité d'apprentissage. Par exemple en utilisant :

```
eta = LearningRate = 0,347867
neigh = Neighboors = [0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000,
                      0.000000, 0.189957, 0.000000, 0.000000, 0.189957, 1.000000,
                      0.000000, 0.000000, 0.000000, 0.189957]
chan = 0.721569
chanArr = [0.959569, 0.884502, 0.873733, 0.842565, 0.832098, 0.760465, 0.770629, 0.746060,
            0.690255, 0.648616, 0.596248, 0.578472, 0.574980, 0.492272, 0.389369, 0.458615]
```

Alors :

```
ResNeigh = [0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000,
            0.000000, 0,066079, 0.000000, 0.000000, 0,066079, 0,347867,
            0.000000, 0.000000, 0.000000, 0,066079]
```

On remarque que la valeur à l'indice correspondant au vecteur vainqueur est égale au maximum de la nouvelle capacité d'apprentissage (34.79%). Les voisins, quant à eux ne pourront apprendre autant de la forme d'entrée. Ils ne pourront en apprendre que 6.61% des données. S'il y avait eu d'autre voisins, plus éloignés du centre du rayon, ils auraient également pu apprendre de la forme d'entrée choisie mais dans une proportion inférieur à 6.61%. Puis on a :

```
ResCharArr = [-0,611702, -0,536635, -0,525866, -0,494698, -0,484231, -0,412598,
              -0,422762, -0,398193, -0,342388, -0,300749, -0,248381, -0,230605,
              -0,227113, -0,144405, -0,041502, -0,110748]
```

Bête soustraction. Et enfin :

```
Res = [0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000,
       0.000000, -0,026312, 0.000000, 0.000000, -0,026312, -0,080220,
       0.000000, 0.000000, 0.000000, -0,0263129]
```

Bête multiplication. Tous les neurones qui ne font pas parti du voisinage et qui ne sont pas vainqueurs ne seront pas modifiés puisque la valeur associée à leur indice dans le vecteur de modification ci-dessus est 0.

Enfin il ne reste plus qu'à appliquer les modifications des neurones en utilisant le vecteur fraîchement calculé avec :

```
/* Update the network weight vectors values */
arr_add(WR, deltaR, nbNeurons);
arr_add(WG, deltaG, nbNeurons);
arr_add(WB, deltaB, nbNeurons);
```

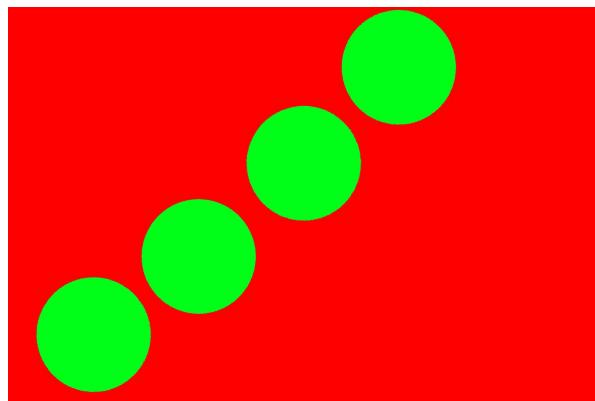
Il suffit en fait d'additionner le vecteur de modification avec le vecteur contenant les valeurs du canal donné et les modifications seront ainsi appliquées aux neurones de la carte. Évidemment, ajouter 0 à une valeur R, G ou B d'un vecteur ne le modifiera pas mais ajouter une valeur positive ou négative modifiera effectivement ce vecteur afin qu'il ressemble d'avantage au vecteur d'entrée choisi.

Il n'y a plus qu'à répéter cette boucle d'apprentissage.

F. Comparaison avec d'autres programmes

Avant d'effectuer les comparaisons entre posternn et d'autres programmes/algorithmes de postérisation je tiens à souligner que posternn fonctionne à l'aide d'un réseau de neurone, plus précisément une carte auto-adaptative (si, si...) et par conséquent ses résultats sont aléatoires et dépendent des formes d'entrée présentées au réseau ainsi que de l'ordre dans lequel elle sont présentées.

Afin de vérifier la sortie des programmes avec lesquels je compare posternn j'utilise un petit script python qui utilise la bibliothèque PIL afin de compter le nombre de couleurs d'une image. Attention l'encodage d'image JPG, au cours de l'enregistrement de l'image, va ajouter énormément de couleurs. Ceci est du à la complexité de l'encodage JPEG (voir <https://en.wikipedia.org/wiki/JPEG#Encoding>) mais cela dépasse un peu le sujet de ce mémoire. Je conseille donc d'utiliser des images au format PNG pour pouvoir compter efficacement leur nombre de couleurs. Plus de détails https://en.wikipedia.org/wiki/Wikipedia:How_to_reduce_colors_for_saving_a_JPEG_as_PNG. Par exemple, prenons l'image magnifique ci-dessous que j'ai créé dans GIMP en utilisant 2 couleurs (#00FF18 et #FF0000) et enregistré au format JPG, qualité maximum.



Une image créée en utilisant deux couleurs et enregistré au format JPG et au format PNG.

Puis lançons mon script qui détecte le nombre de couleurs d'une image (le script python se trouve dans le répertoire "utils") :

```
$ python ./utils/get_img_color_number.py ./tmp/ex.jpg  
30
```

Puis réessayons en enregistrant cette fois-ci l'image en PNG :

```
$ python ./utils/get_img_color_number.py ./tmp/ex.png  
2
```

Sans approfondir, c'est pour cette raison que les tests suivants seront effectués avec des images PNG.

1) Comparaison avec GIMP

GIMP est un programme de traitement d'image dont j'ai déjà parlé dans la section 2. Il est très complet et permet notamment de postériser des images. L'algorithme utilisé pour la postérisation est mentionné et rapidement expliqué dans la section 2.C..

La doc de GIMP dit ceci à propos du filtre de postérisation :

Posterize Levels

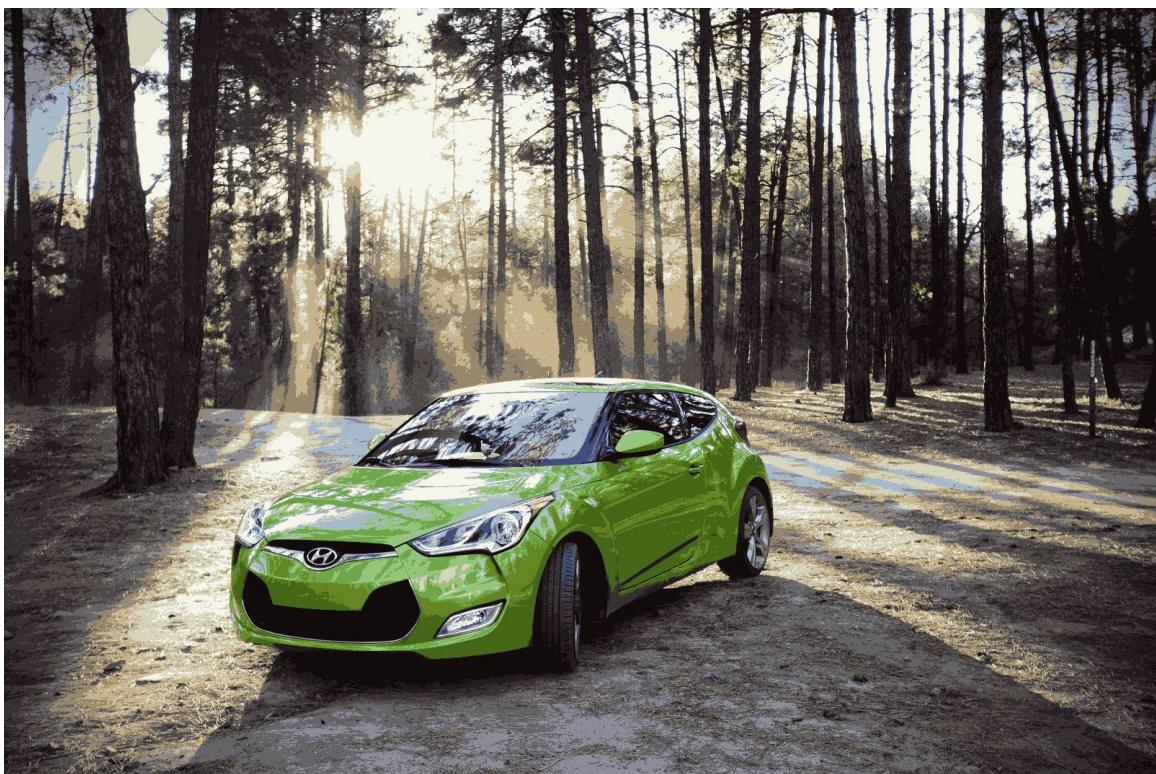
This slider and the input boxes with arrow-heads allow you to set the number of levels (2-256) in each RGB channel that the tool will use to describe the active layer. The total number of colors is the combination of these levels. A level to 3 will give $2^3 = 8$ colors.

Extrait de <http://docs.gimp.org/2.8/en/index.html>

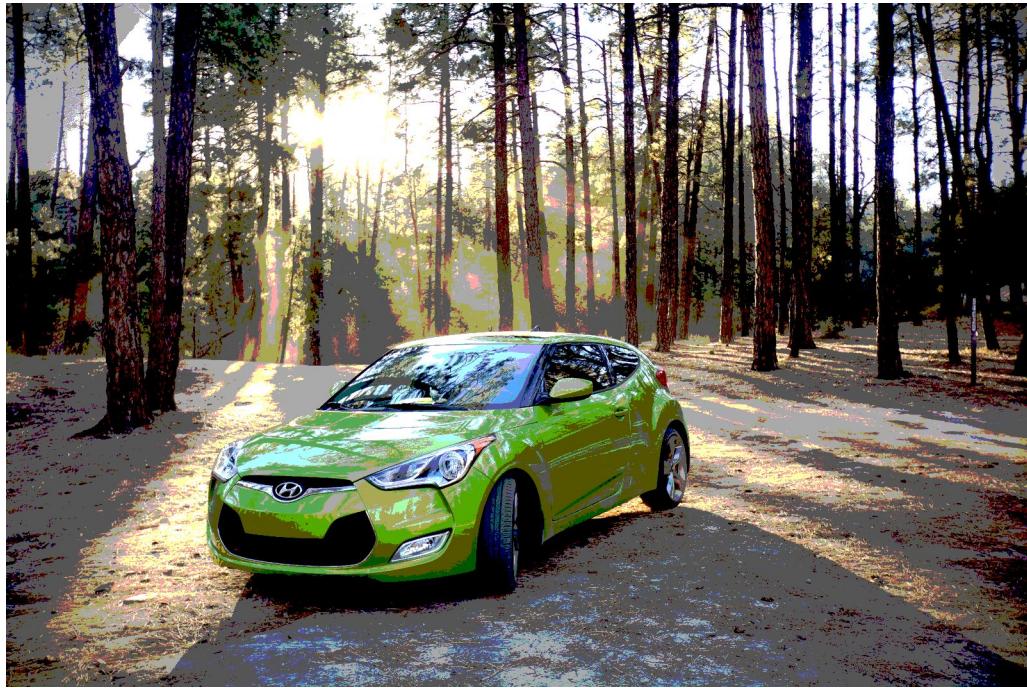
Les niveaux de postérisation ne fonctionnent donc pas tout à fait comme dans mon programme puisque posternn utilisera $3^2 = 9$ couleurs pour un niveau de postérisation de 3. Cependant $4^2 = 2^4 = 16$ donc il est possible de comparer les résultats des deux programmes de façon efficace.

Les deux images suivante présentent le résultat d'une postérisation utilisant 16 couleurs. La première image est obtenue avec posternn et la seconde est le résultat de l'option de postérisation de GIMP.

posternn 16 couleurs



GIMP 16 couleurs



On voit que les couleurs retenues par l'algorithme de GIMP sont plus vives que celles de l'image créée par posternn. Il est difficile de dire quelles couleurs sont mieux choisies mais l'image générée par GIMP semble d'avantage postérisée alors que les deux utilisent en réalité le même nombre de couleurs.

2) Comparaison avec picmonkey

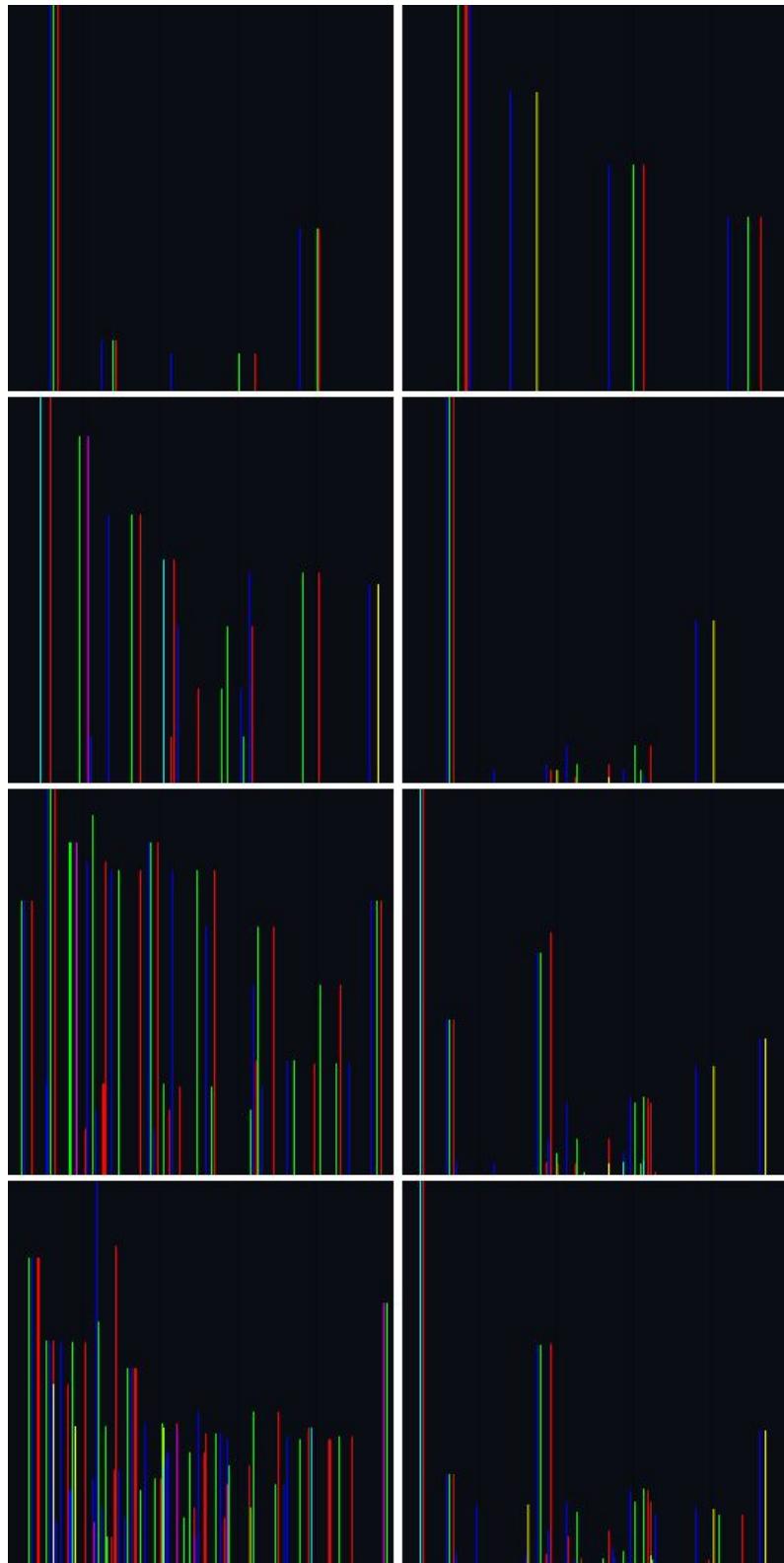
Picmonkey (<http://www.picmonkey.com/>) est un éditeur photo gratuit, avec des options payantes.

C'est un outil pas mal utilisé par les community managers : il permet de créer facilement du contenu visuel très attractif. On peut retoucher des photos ou bien ajouter des effets, on peut notamment postériser une image.

Et la postérisation d'image est bien pensée, elle est surtout paramétrable et on peut choisir le nombre de couleurs entre 2 et 30. Dans cette fourchette posternn peut postériser des images en utilisant 4, 9, 16 ou 25 couleurs. En utilisant mon script pour détecter le nombre de couleurs d'une image j'ai cependant remarqué que (bien que tout soit encoder en PNG), les images enregistrées via picmonkey n'ont pas toujours le nombre exacte de couleurs attendues. Je ne peux toutefois dire à quoi cela est du, le code de picmonkey n'est pas disponible et pour cette même raison je ne sais pas si le service internet n'ajoute pas d'autres effets lors de la postérisation. Cela n'est cependant pas dramatique et permet tout de même de comparer les résultats avec ceux de posternn.

Sur les photos suivantes on voit que les images générées par picmonkey utilisent des couleurs plus sombres que celles générées par posternn mais moins "flashy" que celles générées par GIMP et c'est surtout visible sur les comparaisons des images à 4 et 9 couleurs. De ce fait, les zones de couleurs (les bords), sont assez dissemblables dans les

deux premières comparaisons mais semble très proches dans les deux dernières. On peut comparer la postérisation des deux programmes de façon plus objective en regardant les histogrammes RVB :



Histogrammes des images postérisées avec posternn (gauche) et picmonkey (droite).

Cette image regroupe les histogrammes des images de posternn (à gauche) et de picmonkey (à droite), de 4, 9, 16 et 25 couleurs (de haut en bas).

Il est clair d'après ces histogrammes que picmonkey favorise les couleurs très sombres beaucoup plus que posternn. Les couleurs sont moins contrastées avec posternn. Par contre il y environ le même nombres de couleurs avec les deux programmes, en comptant précisément on s'aperçoit cependant que seul posternn respecte réellement le nombre de couleurs qu'il prétend utiliser pour postériser l'image.

Dans tous les cas les résultats de posternn sont conforme à mes attentes et au but même du projet. Les couleurs sélectionnées par le réseau de neurones sont cohérentes et les différents niveaux de postérisation fonctionnent de la façon attendue.

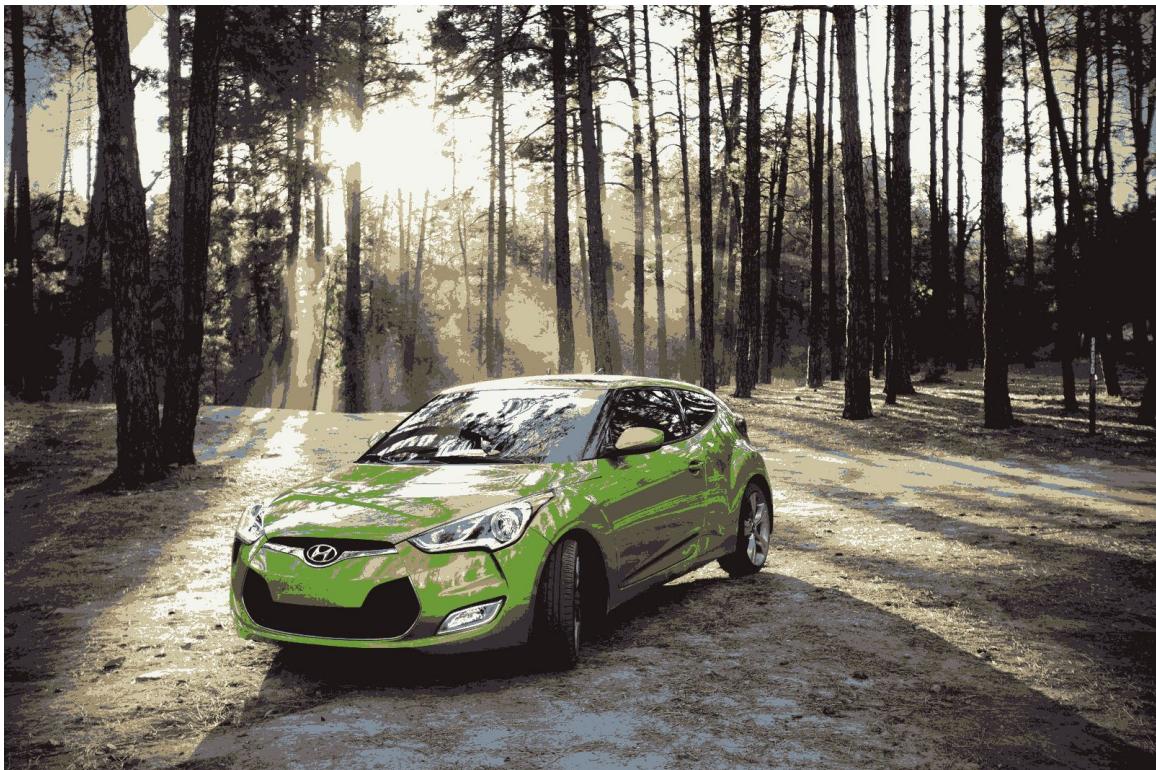
posternn 4 couleurs



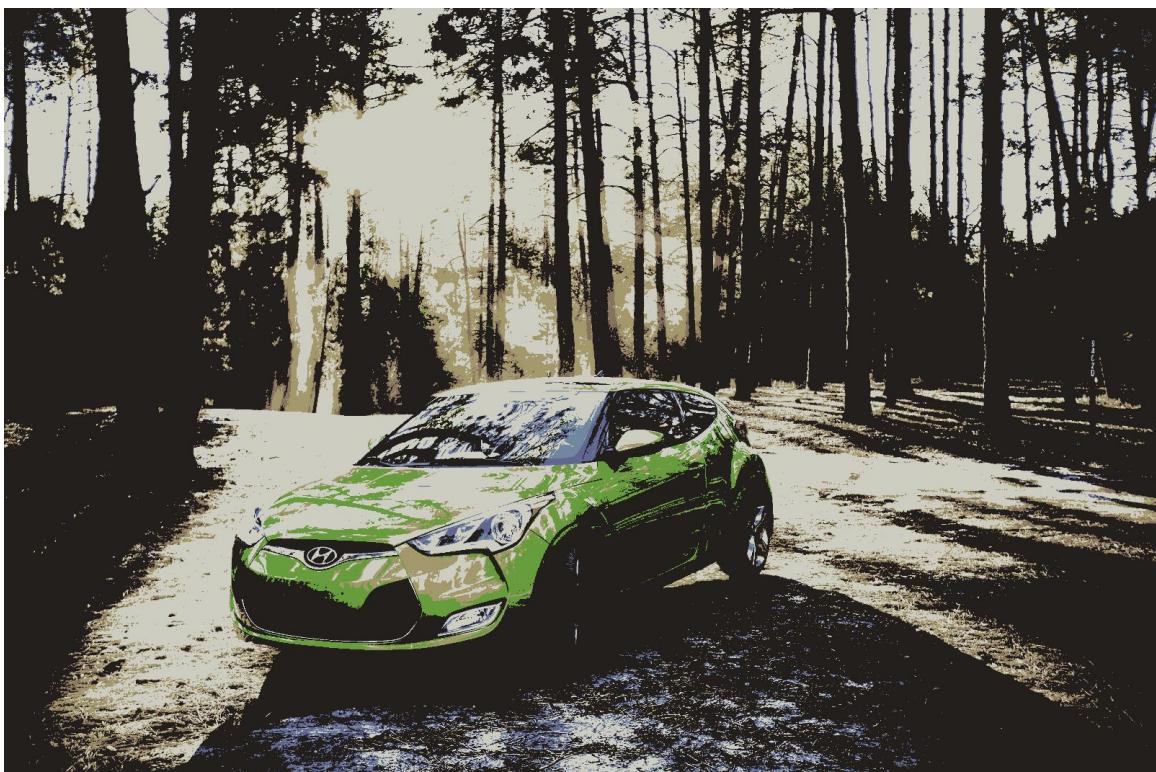
picmonkey 4 couleurs



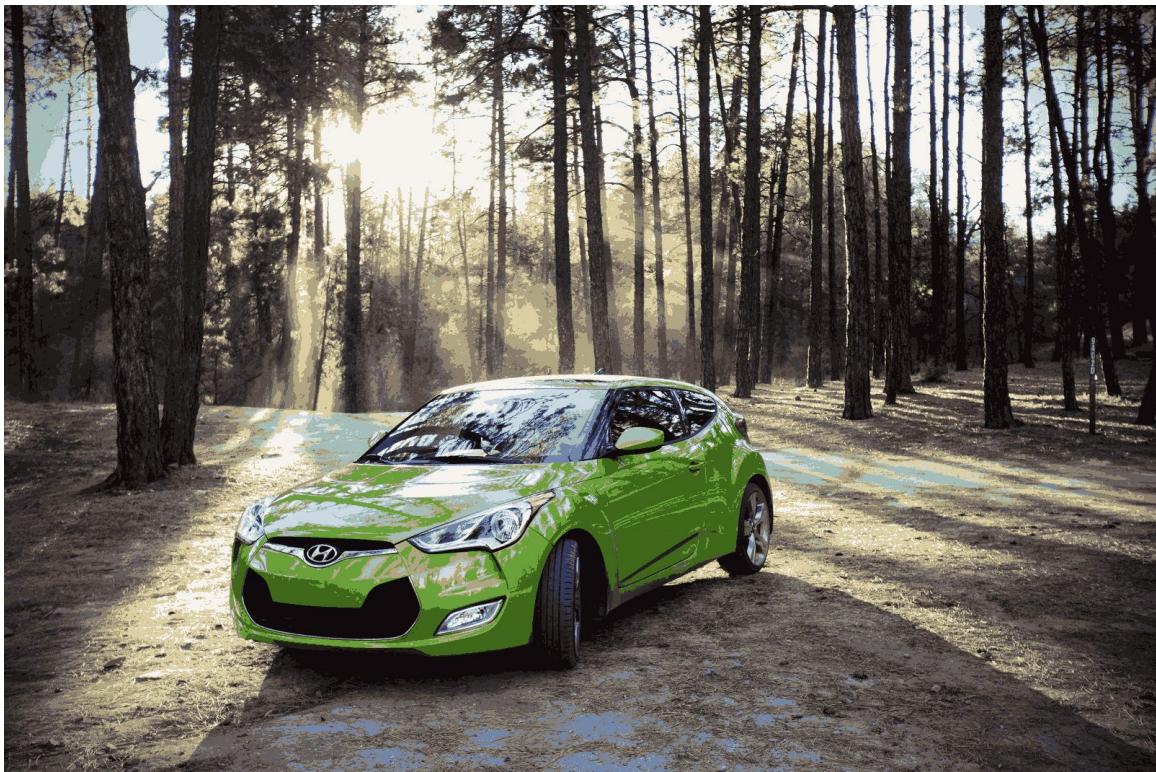
posternn 9 couleurs



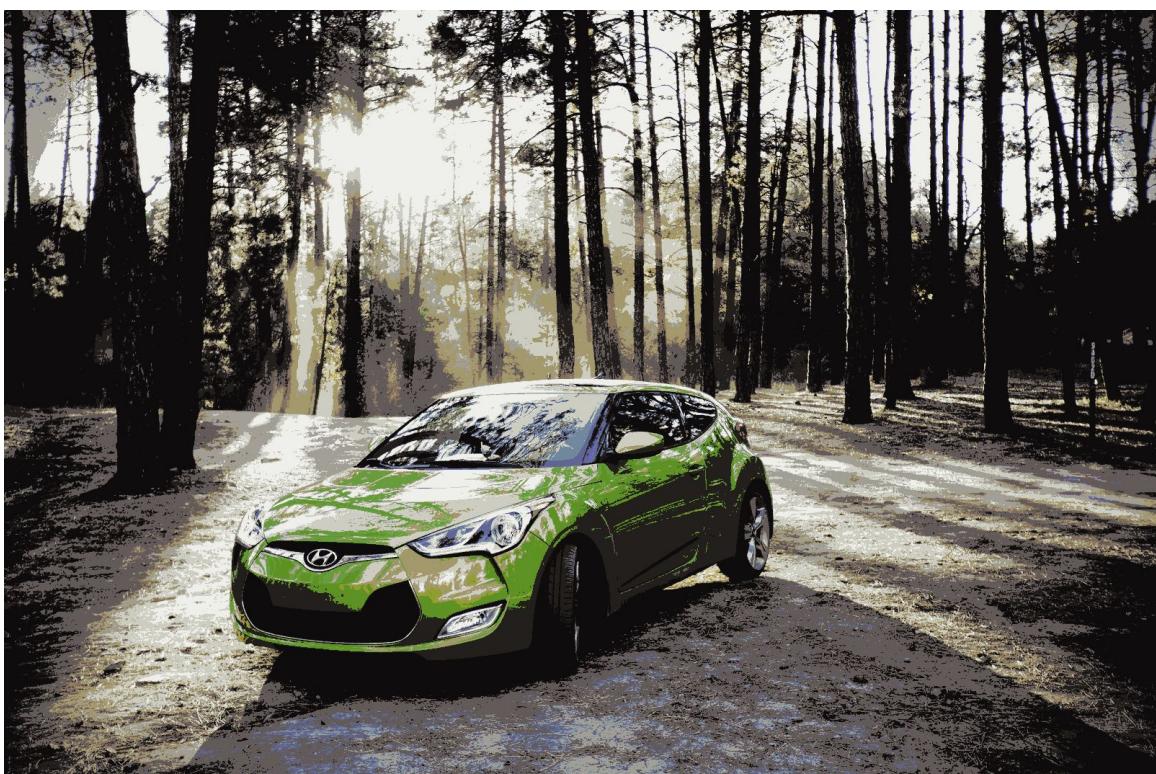
picmonkey 9 couleurs



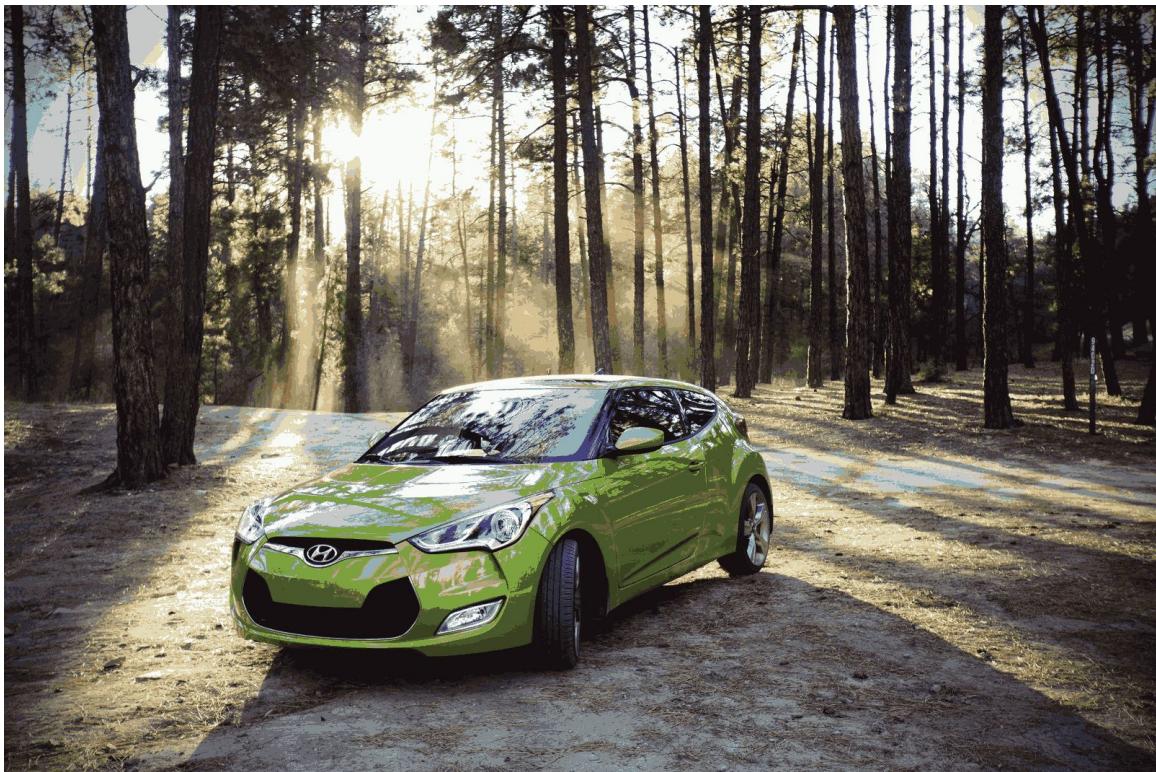
posternn 16 couleurs



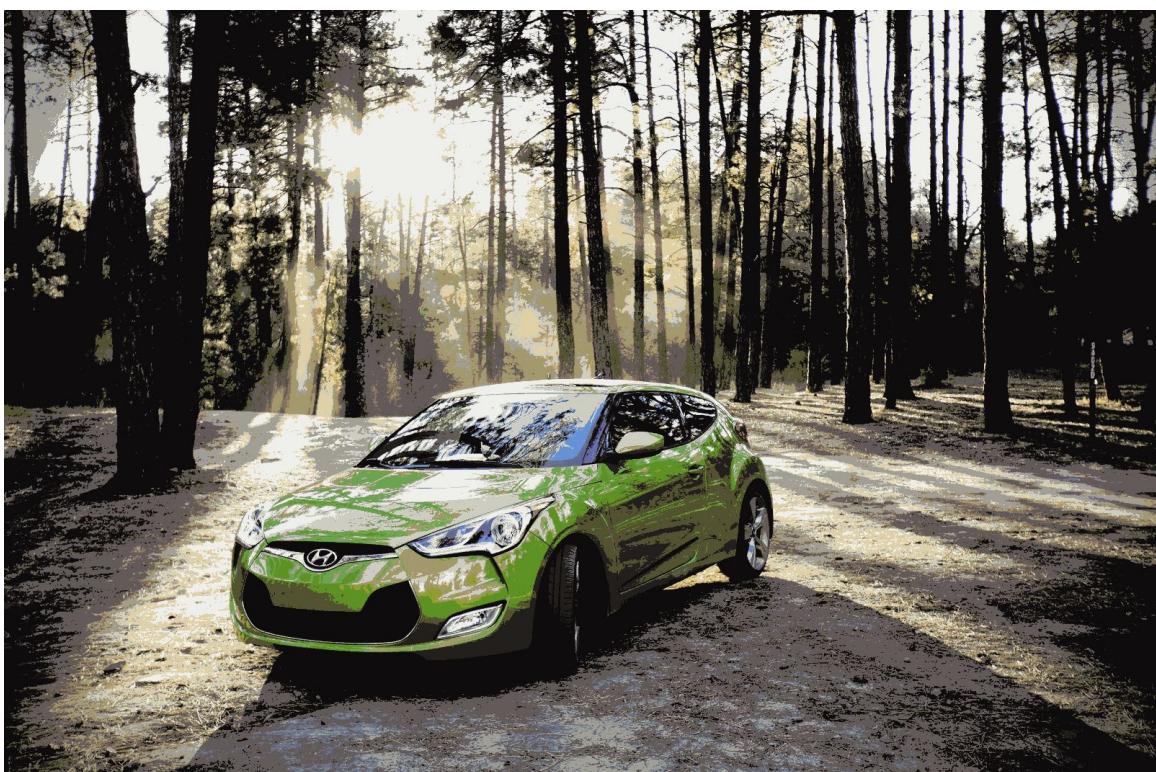
picmonkey 16 couleurs



posternn 25 couleurs



picmonkey 25 couleurs



G. Documentation technique

La programme n'est pas techniquement documenté (quois qu'un peu quand même) dans ce mémoire mais la documentation, à destination des développeurs existe. Elle a été générée avec doxygen à partir des commentaires formatés des fichiers sources. Vous pouvez la trouver dans le répertoire "doc". La page principale de la documentation est "index.html". La page principale propose également une brève description du programme ainsi que quelques exemples concrets d'utilisation.

Une documentation auto-générée ne vaut sûrement pas une documentation manuelle mais je pense que le travail effectué par doxygen est largement suffisant pour ce programme qui n'est pas très compliqué.

5. Conclusions

Le but de ce projet était de démontrer qu'il est possible d'utiliser un réseau de neurones de type carte auto-adaptatives afin de postériser une image en laissant le réseau apprendre à partir des couleurs des pixels de l'image originale et en paramétrant le nombre de couleurs utilisées pour postériser l'image, correspondant au nombre de neurones de la carte. Les résultats devant être cohérents, c'est-à-dire que l'effet apporté à l'image après sa modification par la carte auto-adaptative doit correspondre à une postérisation et non à un autre effet visuel. L'effet doit ainsi s'approcher des effets de postérisation des logiciels de manipulation d'images reconnus et correspondre à une réduction des couleurs de l'image de départ inversement proportionnelle au niveau de postérisation.

En rédigeant ce mémoire j'ai tenté d'exposer ce que j'ai appris sur la postérisation d'images dans un premier temps (section 1.) puis j'ai tenté d'expliquer le fonctionnement des cartes auto-adaptatives en utilisant un exemple concret, à la fois en rapport avec ce projet et également très répandu dans la littérature concernant les cartes auto-adaptatives (section 2.). Puis dans la section 3. je fais le lien entre la postérisation d'images et l'utilisation des cartes auto-adaptatives en exposant le fonctionnement de mon programme.

Je n'ai rencontré aucune difficulté particulière lors de l'écriture du programme, à part peut-être avec les vecteurs C (`malloc()`, `free()`) mais c'est un obstacle quasiment obligatoire lorsqu'on programme en C. J'ajoute que le programme n'a aucune fuite mémoire, si vous testez avec valgrind vous obtiendrez sûrement des erreurs mais elles sont toutes dues à OpenCV, si vous demandez à valgrind de prendre en compte les suppressions spécifiques à OpenCV avec l'option `--suppressions`, vous ne devriez plus avoir une seule erreur. La mise en place de la structure de la carte auto-adaptative et le "codage" de ses algorithmes est très simple car le principe de fonctionnement des SOM est lui-même très simple. Quelques opérations vectorielles et une formule de calcul de distances euclidiennes suffisent presque. La structure a peut-être été un peu plus difficile à mettre en place car j'ai décidé de stocker les données de la carte sous une forme vectorielle quelque peu différente de l'architecture théorique. J'aurais toutefois pu calquer l'architecture du programme sur celle exposée dans la section 2. mais le programme aurait été un peu plus "lourd". En définitive, ce sont les données qui ont été un peu plus compliquées à gérer. En effet, extraire les valeurs de chaque canal de couleur de chaque pixel d'une image n'est pas simple du tout mais heureusement la bibliothèque OpenCV a grandement simplifié le problème. Sa représentation des données extraites ne m'a pas convenue (valeurs dans l'intervalle [0;255], vecteur 1D) mais je n'avais de toute façon pas l'intention de réinventer la roue alors je me suis contenter de formater ces données de façon à ce que le réseau de neurones puisse les assimiler. Pour résumer, l'écriture du code du programme n'a pas posé de problème, excepté pour la transformation des données originales. Et en fait c'est assez logique car dans les réseaux de neurones les algorithmes sont souvent très simples, leur structure n'est pas plus complexe, seul le format des données peut nécessiter d'avantage de réflexion mais ce sont ces données qui dirigent les réseaux au final...

Ce programme est utilisable en l'état bien qu'il ne propose aucune interface, à part la

ligne de commande. Son intérêt, en dehors de ce mémoire est par contre très limité puisque sa seule utilité et la postérisation d'image. Et encore, même pour cette simple utilisation cela reste discutable : les performances sont sûrement moins intéressantes que celles de GIMP ou d'autres logiciels (photoshop, aftershoot, ...) de traitement d'images.

Finalement, ce programme est une sorte de *proof of concept*. Je n'ai, du fait que les logiciels cités précédemment sont souvent payants ou non disponibles pour Linux, pas pu en tester beaucoup. De plus, parmi ceux que j'ai pu tester, seul GIMP est libre et m'a donc autorisé à comprendre le fonctionnement de son algorithme qui n'utilise aucun type de réseau de neurones. Ces réseaux, et surtout les cartes auto-adaptatives, sont souvent utilisés pour traiter des couleurs, notamment pour la reconnaissance de contours [2][5]. Cependant, je n'ai lu aucun article qui traitait de l'utilisation des cartes auto-adaptatives pour la postérisation d'images. posternn démontre parfaitement que cela est possible.

6. Sources

Voici quelques liens qui m'ont été utiles dans mes recherches de documentation et de compréhension.

[1]

<https://docs.google.com/viewer?url=patentimages.storage.googleapis.com/pdfs/US7747100.pdf>

[2] <http://www.cecs.uci.edu/~papers/icme05/defevent/papers/cr1737.pdf>

[3]

<http://cilab.knu.ac.kr/seminar/Seminar/2012/20120804%20False%20contour%20reduction%20using%20neural%20networks%20and%20adaptive%20bi-directional%20smoothing.pdf>

[4] <http://users.ics.aalto.fi/sami/thesis/node23.html#SECTION0007423000000000000000>

[5] https://en.wikipedia.org/wiki/Edge_detection

[6] <http://www.pymvpa.org/examples/som.html>

[7] <http://jjguy.com/som/>

[8] <http://arxiv.org/pdf/1306.3860.pdf>

[9]

<http://www.shy.am/wp-content/uploads/2009/01/kohonen-self-organizing-maps-shyam-guthikonda.pdf>

[10] http://www.novaaims.unl.pt/docentes/vlobo/Publicacoes/2_41_lobo10_IFGIS.pdf