Reconnaissance de l'écriture manuscrite avec SOM
Programmation pour l'intelligence artificielle
Rapport de projet

Edgar Oblette
edwardoblette@gmail.com

–

Étudiant licence informatique
Université Paris VIII
No étudiant: 17812871

Rapport de Projet
11 Mai, 2020

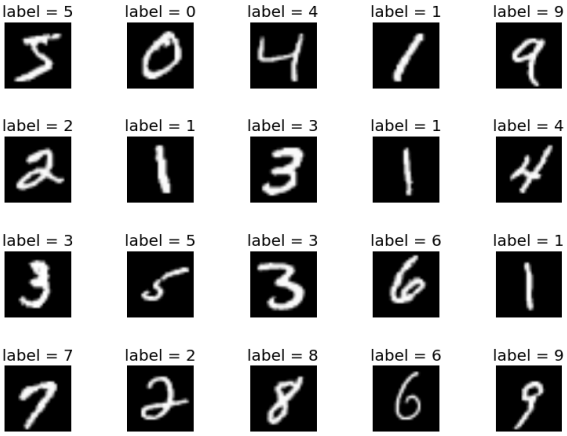
SOMMAIRE

1. Description du projet

Dans le cadre du cours Programmation pour l'intelligence artificielle, nous avions pour projet de réaliser une SOM (Self Organisation Map) qui permettrais de différencier et reconnaitre plusieurs espèces d'Iris. Ce sujet m'a inspiré l'idée de ce projet: réussir à reconnaitre des chiffres écrits de manière manuscrite. J'avais besoin d'un jeu de données pour la phase d'apprentissage, je me suis donc aidé de dataset MNIST. On utilisera le jeu de 60000 images de chiffres manuscrits, et 10000 images pour faire le test d'apprentissage.

1.1. MNIST

Pour commencer avec ce projet la première étape fut d'apprendre à utiliser la base de données MNIST et récupérer les images de dimension 28x28 pixels. Les 4 fichiers permettant l'apprentissage et le test sont :



-train-images-idx3-ubyte: *images pour l'apprentissage*
-train-labels-idx1-ubyte: *labels pour l'apprentissage*
-t10k-images-idx3-ubyte: *images pour le test*
-t10k-labels-idx1-ubyte: *labels pour le test*

http://yann.lecun.com/exdb/mnist/

1.2. SOM

Ici je part sur la même structure de SOM que pour le projet d'Iris, mais chaque vecteur de données (image + label) contiendra 784(28x28) valeurs de 0.0 à 1.0 correspondant à la valeur de chaque pixel de l'image (noir et blanc) et un int correspondant à son label.

Vecteur de données (float) =

Valeurs des pixels			Label
0.0 - 1.0 (0)	....	0.0 - 1.0 (783)	0 - 9
FLOAT			INT

Vecteur de données (unsigned char) =

Valeurs des pixels			Label
0 - 255 (0)	....	0 - 255 (783)	0 - 9
UNSIGNED CHAR			INT

2. Utilisation

2.1. Compilation

Le compilateur utilisé est GCC et la compilation est cross-platform Linux/MacOS.
Pour compiler le projet, 2 options de types sont proposées:

- Unsigned Char
- Float

La principale différence entre ces deux types de compilation est que le projet utilisant les *unsigned char* stocke les poids de vecteur sur un espace plus limité, rendant le programme plus rapide, c'est une petite optimisation.
Par defaut la compilation est faite avec des *unsigned char*.

Compilation avec Unsigned Char:

```
make char
```

Compilation avec Float:

```
make float
```

2.2. Execution

Ici, la commande générique pour exécuter le programme:

```
./main dataset Xsize Ysize Apprentissage1 Apprentissage2
```

Dataset = Nombre de vecteurs à utiliser pour l'apprentissage.
Xsize = Taille X de la matrice 2D.
Ysize = Taille Y de la matrice 2D.
Apprentissage1 = Nombre d'apprentissage pour la première phase (voisinage de 3).
Apprentissage2 = Nombre d'apprentissage pour la deuxième phase (voisinage de 1).

Ici, la commande avancée pour exécuter le programme en mode détaillé:

```
./main dataset Xsize Ysize Apprentissage1 Apprentissage2 -V
Nombré_à_représenter
```

-Verbose = permet d'afficher à chaque étape de l'apprentissage l'évolution du réseau de neurone.
Nombre_à_représenter = permet d'afficher à chaque étape de l'apprentissage la représentation du nombre défini par moyenne de chaque neurones labellisés par ce nombre (par default 0).

3. Paramétrage

3.1. Taille du dataset

Avant de commencer l'apprentissage il faut déterminer la taille du dataset à utiliser, au vu du grand volume de données, un apprentissage sur tout le dataset se révèle très long. Pour l'exemple on utilisera la **⅙** du dataset soit 10000 vecteurs de données.

3.2. Taille du réseau de neurones

La taille du réseau de neurone est définie par la fonction suivante:

$$Nneurones = 5\sqrt{Ndataset}$$

Les neurones seront organisés sur une matrice 2D, elle nous permettra de faire évoluer les neurones de voisinage lors de l'apprentissage. La taille de la matrice est équivalente au nombre de neurones.

4. Initialisation des vecteurs de données

4.1. Allouer la mémoire

On alloue dynamiquement la mémoire pour permettre de stocker les vecteurs de données. Les vecteurs seront stockés dans un tableau de type ***`struct digit`***.

```c
typedef struct digit{
    float * image;   //[SIZE]
    int label;
} digit;

#endif
```

Structure digit
digit.h

```c
void dataAllocate(vector * v, params p){
    // init vector data
    v->data = malloc(p.data_size * sizeof(digit));
    for(size_t i = 0 ; i < p.data_size; i++)
        v->data[i].image = malloc(p.data_img_size * sizeof(float));

    v->average = calloc(p.data_img_size, sizeof(float));
    v->size = p.data_size;
}
```

Fonction d'allocation
dataset.c

4.2. Extraire les données de MNIST

Une fois la mémoire allouée, on va lire dans les fichiers de la base de données MNIST pour récupérer les vecteurs de données. On commence par lire le fichier contenant les images, elles sont stockées pixel après pixel. On découpe le fichier par paquet de 784, chaque découpe représente une image. On convertit chacune des images en float de 0.0 à 1.0 dans le cas de l'implémentation *float* et entre 0 et 255 dans le cas de l'implémentation *unsigned char*.

```c
void read_mnist_char(char *file_path, int num_data, int len_info, int arr_n, unsigned char ** data_char, int info_arr[]){
    int i, fd;
    unsigned char *ptr;

    if ((fd = open(file_path, O_RDONLY)) == -1) {
        fprintf(stderr, "couldn't open image file");
        exit(-1);
    }

    read(fd, info_arr, len_info * sizeof(int));

    // read-in information about size of data
    for (i=0; i<len_info; i++) {
        ptr = (unsigned char *)(info_arr + i);
        FlipLong(ptr);
        ptr = ptr + sizeof(int);
    }

    // read-in mnist numbers (pixels|labels)
    for (i=0; i<num_data; i++) {
        read(fd, data_char[i], arr_n * sizeof(unsigned char));
    }

    close(fd);
}
```

Fonction d'extraction
data/extract.c

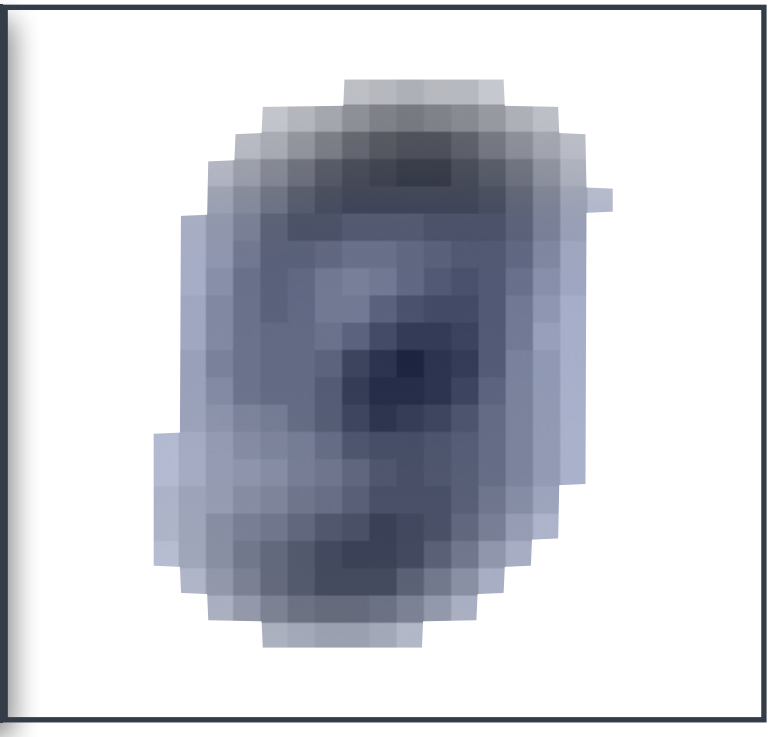
```c
void image_char2float(int num_data, unsigned char * * data_image_char, vector * v){
    int i, j;
    for (i=0; i<num_data; i++)
        for (j=0; j<SIZE; j++)
            v->data[i].image[j]  = (float)data_image_char[i][j] / 255.0;
}


void label_char2int(int num_data, unsigned char ** data_label_char, vector * v){
    int i;
    for (i=0; i<num_data; i++)
        v->data[i].label  = (int)data_label_char[i][0];
}
```

Fonction de conversion
data/extract.c

4.3. Normalisation des données

Les données de la base MNIST sont déjà normalisées, les images résultantes contiennent des niveaux de gris grâce à un technique d'anticrénelage utilisée par l'algorithme de normalisation. Les images ont été centrées dans une image 28x28 en calculant le centre de masse des pixels, et en translatant l'image de manière à positionner ce point au centre du champ 28x28.
Ici la fonction dite de "Normalisation" permet de calculer la moyenne de toute les images permettant par la suite de définir une base d'initialisation pour les neurones.

```c
/* Create an average image from dataset
 * @param vector of data, parameters
 */
void vectorNormalize(vector * v, params parameters){
    size_t image_size = parameters.data_img_size;
    float * average = calloc(image_size, sizeof(float));

    for(size_t data=0; data < v->size; ++data){
        float sum = 0;

        //increment sum for every row
        for(size_t i=0; i < image_size; ++i)
            sum += pow(v->data[data].image[i], 2);

        float norm = sqrt(sum);

        //increment average from every row
        for(size_t i=0; i < image_size; ++i)
            average[i] += (v->data[data].image[i]/ norm);

    }

    //set average from every row
    for(size_t i=0; i < image_size; ++i)
        v->average[i] = (average[i]/ image_size);

    free(average);
}
```



Fonction de normalisation
data/dataset.c

Image moyenne
retour terminal

5. Réseau de neurones

5.1. Initialisation des neurones

L'allocation mémoire pour contenir les neurones a déjà été faite lors de l'allocation des vecteurs de données. Il reste maintenant à les initialiser. On utilise l'image moyenne crée précédemment qu'on va affecter pour chaque neurone en variant légèrement chaque pixel de manière aléatoire entre *-0.05* et *0.025* . On dispose également chaque neurone sur la matrice 2D.

```c
/* Init neuron with normalized data input
 * @param vector of iris and vector of neurons
 */
void vectorNeuralInit(vector * v, neural * n, params parameters ){
    float random;
    int row = 0, column = 0 ;
    for(size_t data=0; data < parameters.neurons_size; ++data, ++column ){
        for(size_t i=0; i < parameters.data_img_size; ++i){
            //init every neuron with (+\-) random value
            random = RandNum(-0.05, 0.025);
            n->neurons[data].image[i] = v->average[i] + random;
        }

        //fill the grid with pointer of neurons
        if(column >= parameters.column){
            ++row ;
            column = 0;
        }
        n->grid[row][column] = &n->neurons[data];
        n->neurons[data].row = row;
        n->neurons[data].column = column;
        n->neurons[data].label = -1;
        n->neurons[data].count = 0;

    }
}
```

Fonction d'initialisation des neurones

neuron/neuralset.c

5.2. Apprentissage

On répète les opérations suivantes sur l'ensemble des vecteurs de données, pendant N itération (nombre d'apprentissage).
Le nombre d'itérations optimales est défini par la formule suivante: $500 * Datasize$.

5.2.1. Best Match Unit

Lors de la phase d'apprentissage, on passe chaque vecteurs de données dans un ordre aléatoire, on calcule la somme de la distance euclidienne de chaque pixel du vecteur de données par chaque pixel de chaque neurone de la matrice. Le neurone ayant la somme la plus courte est appelée BMU (Best Match Unit).

```
match_unit * learning(vector * v , neural * n, int * data_flow, params parameters, int learning, float l_default, int learning_area){

    //parameter vars
    int data_size = parameters.data_size;
    int neurons_size = parameters.neurons_size;
    int image_size = parameters.data_img_size;

    // array of best match unit
    match_unit * bmu =  calloc(parameters.data_size, sizeof(match_unit));
    for(size_t i=0; i < parameters.data_size; ++i)
        bmu[i].best_neuron = -1;

    for(int step=0; step < learning; ++step){
        printf("s:%d\n", step);
        dataRandomize(data_flow, parameters.data_size);
        //for 150 ;
        for(int data = 0; data < data_size; ++data){
            //printf("elapsed %d / %d\n",data, data_size );
            //choose data from dataflow index
            int shuffle_data = data_flow[data];
            //set best match unit

            float lower_match_unit = euclidean_distance(v->data[shuffle_data].image, n->neurons[0].image, image_size); // valeur minimum
            float result;

            for(int neuron = 0; neuron < neurons_size; ++neuron){ //start from 1 because alraeady tested for the minimum

                //printf("Learn s:%d d:%d n:%d\n", step, data, neuron );
                //euclidian distance
                result = euclidean_distance(v->data[shuffle_data].image, n->neurons[neuron].image, image_size);

                // set new best match unit if lower than prevous minimum
                if(result <= lower_match_unit){
                    bmu[shuffle_data].best_neuron =  neuron;
                    lower_match_unit = result;
                }
            }

            neighborhood_improving(v->data[shuffle_data], n, parameters, bmu[shuffle_data].best_neuron, step, learning, l_default, learning_area);
        }

        if(parameters.usage == 1)
            labeling(v , n, data_flow, bmu,  parameters, TRUE);
    }

    return bmu;
}
```

Fonction d'apprentissage
learning.c

5.2.2. Modification du BMU et propagation dans les voisins

Une fois le BMU trouvé pour le vecteur de donné X, on va le modifier pour qu'il tente à se rapprocher de X. L'apprentissage du SOM est dit apprentissage compétitif, ainsi les voisins (neurones proches sur la matrice 2D) bénéficient également de l'apprentissage, mais avec une amplitude qui diminue en fonction du temps et de la distance avec le BMU.

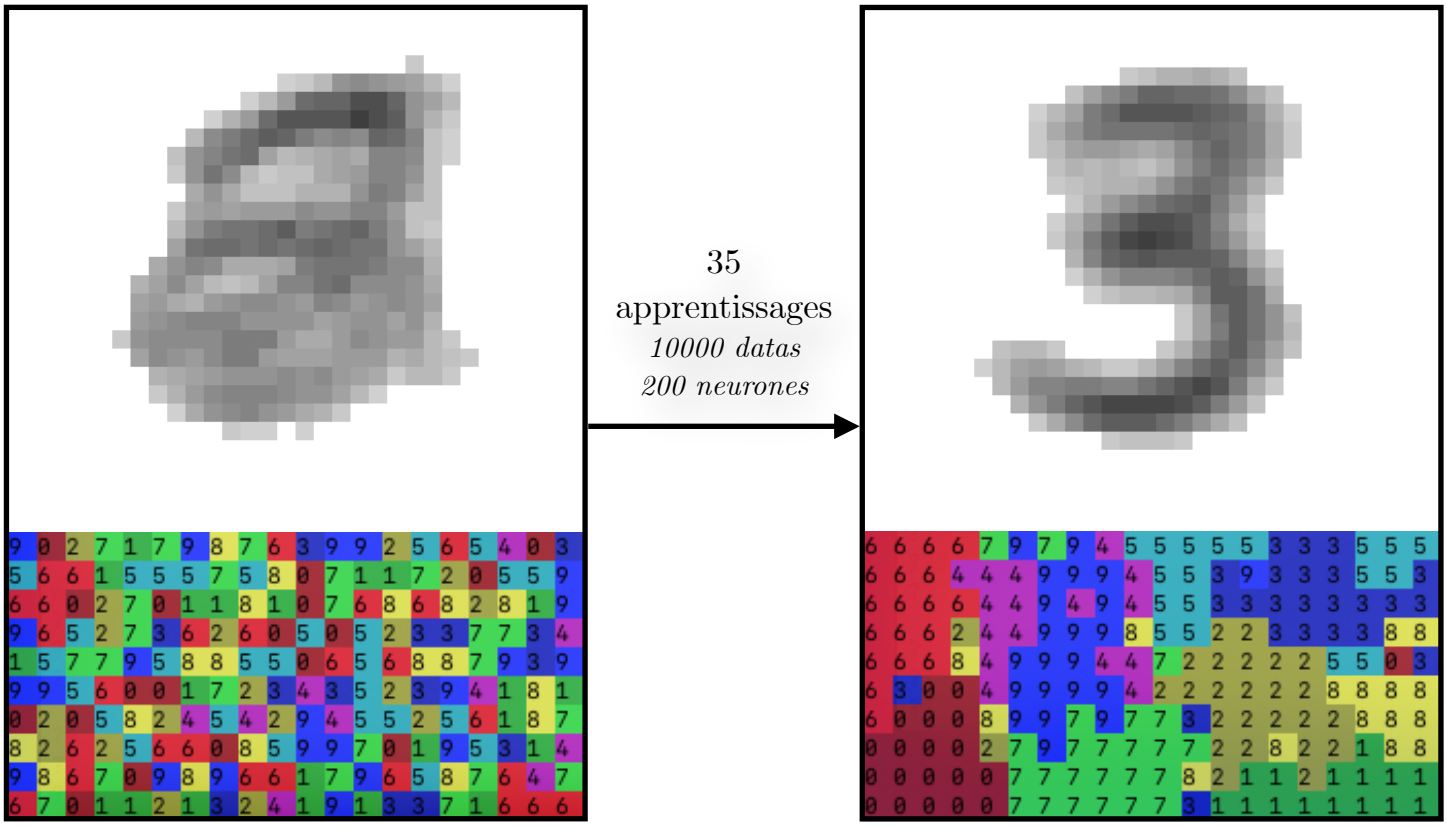
```c
void neighborhood_improving(digit data , neural * n, params parameters, int bmu, int step, int learning, float l_default, int learning_area){

    //parameter var
    int image_size = parameters.data_img_size;
    float learning_rate = l_default * exp (-(float)step/learning);

    //distance var
    int bmu_X =  n->neurons[bmu].row;
    int bmu_Y =  n->neurons[bmu].column;
    int l_a_start = -(learning_area);
    int l_a_end = (learning_area << 1);

    for (int row = l_a_start ; row < l_a_end; ++row){
        for (int column = l_a_start; column < l_a_end; ++column){
            if (((bmu_X + row) >= 0) && ((bmu_X + row) < parameters.row) && ((bmu_Y + column) >= 0) && ((bmu_Y + column) < parameters.column)){
                int neighbor_X = (bmu_X + row);
                int neighbor_Y = (bmu_Y + column);
                neuron * neighbor = n->grid[neighbor_X][neighbor_Y];
                for (int i = 0; i < image_size; i++)
                    neighbor->image[i] += learning_rate * (data.image[i] - neighbor->image[i]);
            }
        }
    }
}
```

Fonction de voisinage

learning.c

5.2.3. Etiquetage

Lorsque la phase d'apprentissage est terminée, il nous reste à étiqueter les neurones pour qu'ils représentent un chiffre (label) lors des tests. Pour cela on détermine pour chaque neurone, quel vecteur de donnée s ressemble le plus au neurone X, le neurone prend ensuite le label du vecteur de données le plus ressemblant.



Représentation d'un neurone en cours d'apprentissage

Représentation du neurone + matrice 2D du réseau neuronal (étiqueté)

6. Test du taux erreur

La fonction de test charge les 10000 vecteurs de données de test présent dans les fichiers *t10k-images-idx3-ubyte* et *t10k-labels-idx1-ubyte*. Le test ressemble à la phase d'apprentissage. On teste pour chaque vecteur de données quel est le neurone le plus proche du vecteur de données, une fois le BMU trouvé, on lui donne le même étiquetage que le BMU et on teste si cet étiquetage correspond à son étiquetage initial. Si oui on incrémente le compteur de test. À la fin du test on calcule le ratio réponse juste sur le total de vecteurs testés.

Apprentissages		Taux d'erreurs pour les vecteurs de données de test				
Phase 1	**Phase 2**	**Dataset**	**1000**	**10000**	**30000**	**60000**
		Neurones	**270**	**500**	**868**	**1225**
0.9 3	**0.9 3**	**Taille de matrice 2D (réseaux de neurones)**	**18 x 15**	**20 x 25**	**28 x 31**	**25 x 49**
5	15		30,043 %	23,1823 %	15,6415 %	14,9714 %
50	150		27,6221 %	21,6421 %	17,5835 %	
500	1500		29,5629 %	21,5621 %		

Résultats des taux d'erreurs
Images test

Les résultats montrent un taux d'erreurs de 15 à 30 % sur le dataset de test (données non apprises). Au vu du très long temps de calcul lors de la phase d'apprentissage, il ne m'est pas possible de réaliser un nombre suffisant d'apprentissages qui permettrait une possible diminution du taux d'erreur.

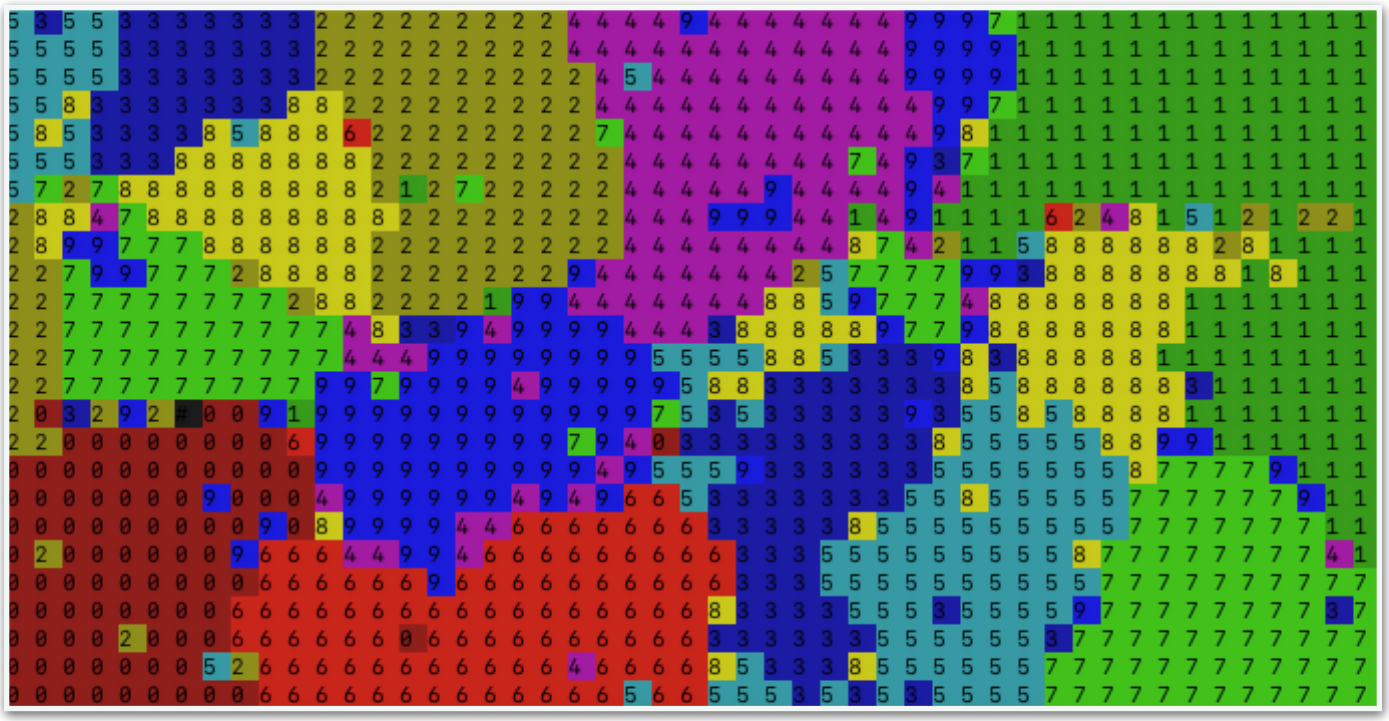
7. Réflexions et améliorations

J'ai réfléchi à différentes pistes d'améliorations lors de ce projet pour obtenir de meilleurs résultats comme réduire la taille des images vers 14 x 14 ou 7 x 7, utiliser une matrice hexagonale et/ou toroïdale, réaliser plusieurs couches de réseaux de neurones. Lors de rédaction de ce rapport j'ai pu améliorer le temps de calcul en stockant les poids de vecteurs de données (784 pixels de chaque images) dans des *unsigned char* à la place de *float*. En effet les *unsigned char* permettent de stocker des valeurs entre 0 et 255 sur un espace de 1 octet seulement, en comparaison des 4 octets utilisé par le *float*.

8. Synthèse

Le SOM arrive à plutôt bien reconnaitre les chiffres manuscrits de la base MNIST, malgré un long temps d'apprentissage. Les phases d'apprentissage pourraient être améliorées en trouvant le bon réglage pour la propagation des voisins. Ici le nombre d'apprentissage est très limité à cause d'un temps de calcul très conséquent. Pour les 60000 vecteurs il aurait fallu théoriquement 30 millions d'apprentissages pour avoir un résultat optimal. Ce qui m'est impossible de réaliser avec ma machine, mais des optimisations comme réduire la taille des vecteurs seraient intéressantes à tester. Le SOM reste limité dans mon exemple par un temps de calcul excessif. L'utilisation de réseau neuronal convolutif me paraît plus indiquée pour ce genre de cas d'utilisation. Car ils sont particulièrement utilisés pour de la reconnaissance d'image et vidéo. En effet, les CNN permettent de mieux gérer des images de grande taille car leurs fonctionnements visent à réduire le nombre d'entrées en appliquant une suite de produits de convolution sur l'image et, entre autres occuper moins d'espace tout en conservant une forte corrélation avec les images d'entrées.

Néanmoins l'implémentation du SOM pour réaliser un système simple de reconnaissance de caractère manuscrit, se révèle être un exercice intéressant et une bonne introduction aux domaines et applications de l'intelligence artificielle.



Réseau de 1225 neurones étiquetés

Apprentissage sur 60000 vecteurs

9. Remerciements

Je tiens tout particulièrement à remercier Monsieur Jean Jacques Mariage pour son enseignement tout au long de l'année et son aide lors de la réalisation de ce projet.

10. Sources

- *http://yann.lecun.com/exdb/mnist/*
- *https://fr.wikipedia.org/wiki/Base_de_donn%C3%A9es_MNIST*
- *https://pubmed.ncbi.nlm.nih.gov/19643706/*
- *http://www.magnusjohnsson.se/Papers/SAIS2010.pdf*