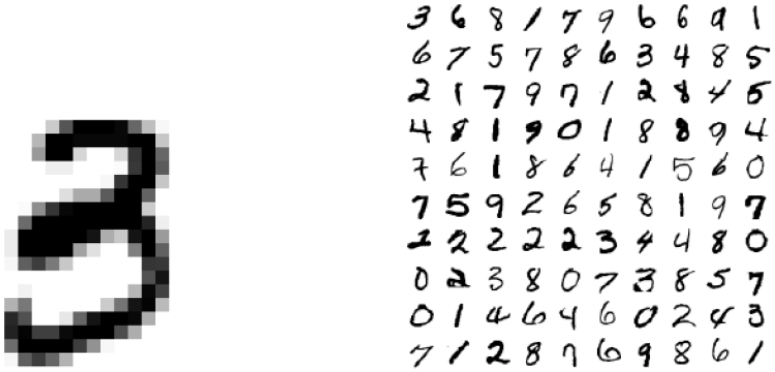
Reconnaissance d'écriture de chiffres et accélération à l'aide d'un kd-arbre

Ensimag 1A - Préparation au Projet C

1 Présentation

L'objet de ce projet est de réaliser un petit programme de reconnaissance d'écriture de chiffres. Le programme que vous allez écrire prendra une ou plusieurs images en entrée, représentant un chiffre dessiné par l'utilisateur. En sortie, votre programme affichera le chiffre reconnu dans chaque image donnée, après en avoir effectué l'analyse.



(a) (b)

FIGURE 1 – (a) Image en niveau de gris de 28×28 pixels, représentant le chiffre 3. (b) Quelques exemples d'images de chiffres issus de la base MNIST.

Comme illustré en figure 1(a), l'image d'un chiffre fourni à votre programme est composée d'un rectangle de pixels, chacun représenté par une valeur entre 0 et 255 encodant son niveau de gris. Cette image n'est donc pas trivialement interprétable par le programme, pour lequel elle est un tableau de valeurs brutes. Pour trouver le chiffre représenté, il faut faire appel à un algorithme de *reconnaissance de motif*. Il existe de nombreux algorithmes de reconnaissance ; une majorité d'entre eux utilise une *base d'apprentissage*, c'est à dire une liste d'images exemples qui ont préalablement été étiquetées par un ou plusieurs experts (ici humain). La figure 1(b) illustre quelques éléments de la base de chiffres MNIST [1], obtenue en faisant écrire les 10 chiffres à plusieurs milliers de personnes. L'expert étiquette chaque image, c'est à dire donne, pour chaque image de la base, le chiffre correspondant à son interprétation de l'image. Le programme de reconnaissance fournit alors pour toute nouvelle image *test* (ne figurant pas dans la base), une étiquette en extrapolant à partir des exemples de la base.

Parmi les nombreux algorithmes de reconnaissance de motifs existant, il en existe un très simple, qui fait l'objet de ce projet. Il s'agit de l'algorithme de *recherche du plus proche voisin* (Nearest Neighbor search). Celui-ci consiste, pour une image test donnée, à trouver la plus proche dans la base d'exemples étiquetés, et à renvoyer l'étiquette correspondante associée dans la base. Cette recherche du plus proche voisin peut être naïve (exhaustive), auquel cas l'image test est comparée à toutes les images de la base d'apprentissage. Mais ceci n'est pas très efficace, particulièrement si l'on souhaite reconnaître un grand nombre de chiffres. C'est pourquoi on précalcule en général une structure intermédiaire facilitant la recherche et permettant d'éviter le parcours exhaustif. Une des structures les plus utilisées pour accélérer cette recherche est le kd-arbre[2]. Vous allez implémenter les structures et algorithmes nécessaires à cette recherche, détaillés ci-dessous.

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

2 Distance entre images

Pour trouver l'image de la base la plus « proche » d'une image test donnée, il faut définir une distance entre deux images. Pour cela on considère que chaque image d'un chiffre est centrée correctement sur le chiffre, et d'une résolution carrée d de taille raisonnable. C'est le cas des images de la base MNIST, qui font 28×28 pixels. Une fois centrées, comparer deux images se ramène à comparer deux à deux leurs pixels. On peut alors traiter chaque image de taille $d \times d$ comme un grand vecteur de valeurs en niveau de gris, de dimension d^2. En notant une image $\mathcal{I}$, une distance $d(\mathcal{I}_1, \mathcal{I}_2)$ peut alors être définie sur la base des normes L_p classiques sur les vecteurs :

$$D_p(\mathcal{I}_1, \mathcal{I}_2) = \left(\sum_{x \in \{1, \cdots, d^2\}} |\mathcal{I}_1(x) - \mathcal{I}_2(x)|^p \right)^{\frac{1}{p}}, \tag{1}$$

où $\mathcal{I}_i(x)$ désigne le niveau de gris du pixel x. En particulier, par exemple, la norme euclidienne entre deux images est très utilisée ($p = 2$). Le coût de calcul de cette distances est en $\mathcal{O}(d^2)$, c'est pourquoi, avant de traiter les images, on en réduit en général la dimension, en faisant une mise à l'échelle de l'image à une dimension plus petite : typiquement $d = 10$. Une fois la distance choisie, et la base d'apprentissage chargée, l'algorithme du plus proche voisin dans sa version naïve/exhaustive est très simple à écrire. Reconnaître une image test consiste en une simple boucle calculant sa distance à chaque image exemple en gardant l'image la plus proche.

Implémentez l'algorithme du plus proche voisin version naïve, en vous appuyant sur les structures de données et fonctions fournies pour le chargement d'images (cf. Annexe). La distance euclidienne entre deux images est fournie.

3 Accélération avec un kd-arbre

La complexité de la reconnaissance en version naïve est en $\mathcal{O}(Nd^2)$ avec N le nombre d'éléments dans la base d'apprentissage, et $\mathcal{O}(PNd^2)$ si l'on veut reconnaître P images tests. Le *kd-arbre* [2, 1] est un arbre binaire permettant de réduire cette complexité. Son but est de partitionner l'espace des images en sous-espaces plus homogènes. La recherche consistera, pour une image test donnée, à identifier le sous-espace approprié pour cette image : ce sous-espace contient probablement ses plus proches voisins, il suffira donc de les identifier pour interpréter l'image test. Bien que cette stratégie ne garantisse pas de trouver la solution optimale, elle donne de bons résultats en pratique.

Le kd-arbre définit un partionnement hiérarchique, dont le principe est une généralisation de la structure d'arbre binaire de recherche, pour des données à plusieurs dimensions. Chaque nœud interne de l'arbre partitionne l'ensemble des images en deux sous-ensembles de taille équivalente, selon la valeur d'un pixel choisi. Ainsi, lors d'une recherche du plus proche voisin d'une image, on pourra itérativement descendre dans l'arbre en explorant l'un ou l'autre des sous-arbres fils en fonction de la valeur dans l'image du pixel choisi, et obtenir un comportement logarithmique.

Dans notre cas, pour obtenir des zones de recherche raisonnables, on arrête le partionnement à une certaine profondeur à la construction, de sorte que les feuilles de l'arbre contiennent au plus $taille\_max$ images candidates. La recherche consistera donc à descendre dans l'arbre jusqu'à atteindre les feuilles, puis comparer l'image à toutes les images de la feuille trouvée. Un exemple complet d'un tel arbre est illustré en figure 2.

En résumé, chaque feuille de l'arbre stocke un ensemble d'images, et chaque nœud interne de l'arbre stocke donc les informations suivantes (voir structures de données fournies en Annexe) :

- L'indice du *pixel séparateur* x_s.
- Un pointeur sur l'image séparatrice $\mathcal{I}_s$ stockée dans le nœud (de type `uint8_t*`, pointeur sur un tableau de pixels niveau de gris `uint8_t`)
- Les pointeurs sur les racines des sous arbres gauches et droits. Toute image $\mathcal{I}$ du sous-arbre à gauche est telle que $\mathcal{I}(x_s) < \mathcal{I}_s(x_s)$, c'est à dire que le niveau de gris du pixel séparateur de $\mathcal{I}$ est inférieur à celui du pixel séparateur de l'image séparatrice. Réciproquement toute image $\mathcal{I}$ du sous-arbre droit est telle que $\mathcal{I}(x_s) \geq \mathcal{I}_s(x_s)$.

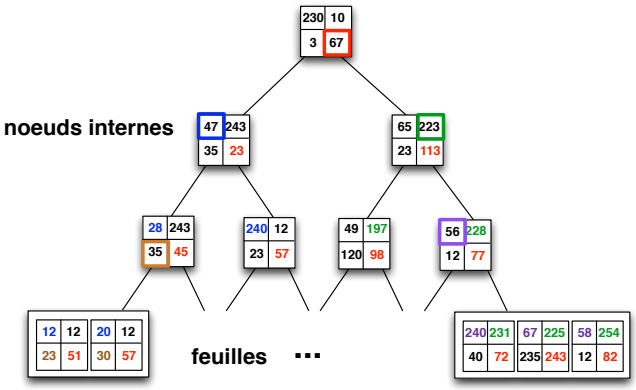


FIGURE 2 – Exemple de kd-arbre à 3 niveaux. L'arbre est représenté avec des images de taille 2×2 (4 pixels), représentées ici en chaque nœud et dans les feuilles. Le carré de couleur représente le pixel séparateur à un nœud interne donné. L'ensemble des images stockées à gauche ont un pixel de valeur inférieure, celles stockées à droite un pixel de valeur supérieure. En particulier, les feuilles regroupent un ensemble d'images respectant donc toutes les propriétés résultant des comparaisons à chaque noeud dont il descend.

3.1 Construction d'un kd-arbre

La construction d'un kd-arbre est récursive. On donne pour la construction d'un kd-arbre le pseudo-code ci-dessous. La fonction de construction d'un arbre prend en entrée l'ensemble des images à structurer. Une bonne construction doit scinder cet ensemble d'images en deux sous-ensembles de taille équivalente pour obtenir un arbre équilibré. Les deux sous-arbres gauche et droit sont ensuite construits par récursion sur ces sous-ensembles.

Algorithm 1 Creation d'un kd-arbre

function CREER_NOEUD(e) ▷ créer un noeud à partir d'un ensemble d'images e
  if $|e| < taille\_max$ **then** ▷ test sur la taille de e, typiquement $taille\_max \sim 1000$
    return struct Feuille $\{e\}$ ▷ structure feuille contenant l'ensemble des images
  else
    $x_s \leftarrow$ SELECTIONNER_PIXEL_SEPARATEUR(e)
    $t \leftarrow$ TRIER(e, x_s) ▷ trier e selon les valeurs au pixel d'index x_s
    $e_g, \mathcal{I}_s, e_d \leftarrow$ SEPARER_MEDIAN(t) ▷ L'image mediane $\mathcal{I}_s$ separe t en e_g, e_d
    return struct Noeud_Interne$\{x_s, \mathcal{I}_s,$ CREER_NOEUD(e_g), CREER_NOEUD(e_d)$\}$
  end if
end function

Représentation des ensembles d'images. Pour éviter toute duplication des données images lors de la manipulation d'ensembles d'images, il est préconisé d'utiliser un tableau de pointeurs d'images (`uint8_t*[]` ou de manière équivalente `uint8_t**`) pour les représenter. La manipulation de l'ensemble d'images ne nécessite alors aucune copie d'image.

Sélection du pixel séparateur. Un critère naïf facile à implémenter dans un premier temps, est de sélectionner un pixel séparateur aléatoirement. Mais il est peu efficace pour la recherche. Un critère très efficace est de choisir un pixel avec la plus large plage de valeurs sur l'ensemble des images à partitionner : cela permet de sélectionner un pixel qui sera discriminant pour la recherche. Pour le mettre en oeuvre, on pourra calculer la différence entre max et min des valeurs de chaque pixel sur l'ensemble des images, pour conserver celui aux valeurs les plus étendues.

Sélection de l'image séparatrice. L'algorithme de construction sélectionne l'image permettant d'obtenir deux sous-ensembles de taille équivalente. Pour cela, il trie l'ensemble des images selon la valeur (niveau de gris) du pixel séparateur, pour enfin stocker l'image médiane de l'ensemble dans le nœud. On pourra utiliser `qsort` (l'implémentation de quicksort de la bibliothèque standard C) pour le tri.

Représentation des noeuds. Les nœuds peuvent être de deux types, noeud interne ou feuille. On impose pour cela une structure de données basée sur une union, permettant de stocker les deux types de nœuds de l'arbre dans un même espace mémoire (cf. Annexe).

3.2 Recherche du plus proche voisin dans un kd-arbre

Pour trouver le plus proche voisin d'une image test, on examine les nœuds internes en descendant jusqu'à trouver la feuille dont la cellule contient l'image test, puis on compare l'image test aux images de cette feuille pour ne retenir que la plus proche au sens de la distance considérée. Cette stratégie, bien qu'approximative (ne garantit pas de trouver le plus proche voisin de tout l'ensemble initial, qui pourrait se trouver dans une autre feuille voisine...), donne de bons résultats en pratique avec des valeurs de $taille\_max$ suffisamment grandes, $taille\_max \sim 1000$.

Algorithm 2 Recherche dans un kd-arbre a de l'image $\mathcal{I}_c$ la plus proche de $\mathcal{I}$

```
1:  function APPROX_NN(a,I,DISTANCE)   ▷ voisin de l'image I dans l'arbre a au sens de DISTANCE
2:      n ← NOEUD_RACINE(a)                                              ▷ noeud courant
3:      while TYPE(n) = NOEUD_INTERNE do
4:          I_c ← MIN_DISTANCE_I({I_c,I_s})      ▷ comparer candidat I_c à l'image séparatrice I_s de n
5:          if I(x_s) < I_s(x_s) then            ▷ descendre selon la valeur du pixel separateur x_s de n
6:              n ← FILS_GAUCHE(n)
7:          else n ←FILS_DROIT(n)
8:          end if
9:      end while
10:     I_c ← MIN_DISTANCE_I({I_c}∪ IMAGES_FEUILLE(n))   ▷ on garde l'image I_c la plus proche de I
11:     return I_c
12: end function
```

où $\text{MIN\_DISTANCE}_{\mathcal{I}}(\mathcal{E})$ renvoie l'image de l'ensemble $\mathcal{E}$ la plus proche de $\mathcal{I}$ au sens de DISTANCE. La fonction APPROX_NN ci-dessus renvoie l'image la plus proche trouvée, il vous faudra encore trouver l'étiquette correspondante en utilisant la fonction `get_index` du module `array`, qui permet à partir d'un pointeur sur une image de retrouver son indice dans l'array correspondant.

4 Travail demandé

Pour le mini-projet, il est demandé d'implémenter les modules C permettant la construction d'un kd-arbre et la recherche d'une image à partir de cette structure de données. Un programme principal permettra de valider le bon fonctionnement de votre projet, dont les spécifications complètes se trouvent en annexe de ce document.

4.1 Modules attendus

— module `kdtree.c` : fonctions `create_kdtree` et `free_kdtree`, respectivement construction et désallocation d'un kd-arbre. Un binaire du module `kdtree.o` vous est fourni dans un premier temps pour que vous puissiez tester vos algorithmes de recherche, à substituer par votre implémentation.

— module `NN.c` : fonctions `brute_force_NN` et `kdtree_approx_NN` de recherche de plus proche voisin pour un ensemble d'images test, respectivement sans et avec l'aide d'un kd-arbre. Un binaire du module `NN.o` vous est fourni dans un premier temps pour que vous puissiez tester votre module kdtree.c de manière indépendante, et que vous puissiez comparer les résultats d'une recherche effectuée par votre module NN.c avec celui distribué par les enseignants.

4.2 Progamme attendu

Nous souhaitons trouver dans l'archive un programme `recognize`, qui prend en entrée une seule image de type pgm et renvoie sur la sortie standard uniquement le chiffre reconnu. Les arguments seront fournis via ligne de commande. Les options en ligne de commande devront permettre de spécifier si l'on

applique l'algorithme naïf ou le kd-arbre, ainsi que la dimension d à laquelle les images seront réduites pour la reconnaissance, et $taille\_max$ dans le cas du kd-tree. Un squelette de code `recognize.c` à compléter vous est fourni pour démarrer.

4.3 Tests

Vous écrirez et fournirez tout test que vous jugerez pertinent permettant de vérifier la correction du programme. Le programme devra être exempt de fuite mémoire et être débogué. Pour contrôler le bon fonctionnement du module, il est conseillé de :

- comparer l'exécution à celle de l'algorithme naïf.
- contrôler le temps d'éxécution du programme avec la commande UNIX `time`
- compter le nombre de nœuds visités et le nombre de distances image calculées dans le cadre des recherches dans l'arbre

4.4 Approfondissement facultatifs

Performances (*). En extension, vous pouvez implémenter un programme `perf` qui teste les images de la base de test MNIST (voir annexe) et renvoie sur la sortie standard le taux de reconnaissance (nombre d'images correctement reconnues / nombre d'image tests total). Les options en ligne de commande devront permettre de spécifier si l'on applique l'algorithme naïf ou le kd-arbre, la dimension d, mais aussi le nombre d'images testées de la base MNIST (on évitera de tester systématiquement l'ensemble complet des 10'000 images tests, ce qui prend plusieurs minutes). Préférer travailler avec un sous ensemble de 10, 100 ou 1'000 images tests pour le débogage. On utilisera pour cela la fonction `resize_array` du module array (cf. Annexe).

Parcours au sens des n premiers candidats (**). Une variante est de rechercher le plus proche voisin dans l'arbre parmis les n plus proches candidats : avec un ensemble de feuilles plus petites, la recherche n'est plus une simple descente de l'arbre vers la meilleure feuille mais une descente en profondeur d'abord, où l'on décide d'explorer éventuellement l'autre fils si le nombre de candidats examinés est inférieur à n.

5 Annexe

5.1 Spécifications fournies, à implémenter

Module kdtree. Une structure de données vous est imposée pour le kd-arbre, permettant la représentation des types de nœuds sous forme d'une union `node` de deux structures (interne : `kdbranch`, feuille : `kdleaf`). La caractérisation du nœud repose sur un type enuméré `kdnode_type`, et l'accès à l'une ou l'autre des interprétations du nœud se fera en fonction, avec le champs leaf et branch de l'union. Chaque image est supposée stockée comme un tableau de d^2 pixels, indexable de 0 à $d^2 - 1$ (une image est donc de type `uint8_t*`).

```
/* pre-declaration du type noeud du kdtree */
struct kdnode;

/* les noeuds du kdtree peuvent etre de 2 types: interne ou feuille */
/* noeud interne */
struct kdbranch {

        const uint8_t *split_image;   /* pointeur sur l'image separatrice */
        uint32_t split_pixel;         /* indice du pixel separateur */

        /* pointeur vers les sous-arbres divises selon la valeur du
         * pixel "split_pixel" de l'image separatrice */
        struct kdnode *left, *right;
};

/* feuille */
```

```
struct kdleaf {
        /* ensemble d'images de la feuille (tableau de pointeurs vers
         * les donnees de chaque image) */
        const uint8_t **images;

        /* nombre d'images de la table se rapportant a cette feuille */
        uint32_t size;
};

/* type enumere permettant de caracteriser le type d'un noeud kdnode */
typedef enum {
        BRANCH,                 /* caracterisation noeud interne */
        LEAF                    /* caracterisation feuille */
} kdnode_type;

/* structure de noeud du kd-arbre */
struct kdnode {
        kdnode_type type;
        union {
                struct kdleaf leaf;
                struct kdbranch branch;
        } node;
};

/* structure encapsulant un kd-arbre */
struct kdtree {
        const struct array *dataset;  /* pointeur sur images de la base*/
        struct kdnode *root;          /* racine du kd-tree */
        uint32_t nb_pixel;            /* nb de pixels d'une image*/
        uint32_t linear_cutoff;       /* taille max des feuilles */
};


/* functions pour creer et detruire un kdtree */

/* creer un kdtree a partir d'un jeu de donnees. NOTE: le kd-tree peut
 * stocker des pointeurs sur des images du jeu de donnees et donc
 * l'utilisation du kdtree suppose que le dataset associe n'est pas
 * modifie ou desalloue */
struct kdtree* create_kdtree(const struct array* dataset, uint32_t linear_cutoff);

/* libere toute la memoire du kdtree */
void free_kdtree(struct kdtree*);
```

Module NN. Le module NN declare deux fonctions de recherche de plus proche voisin, à implémenter dans `NN.c`.

```
/* recherche du plus proche voisin au sens de la distance "dist_func"
 * pour un groupe d'image tests, par l'algorithme naif exhaustif. La
 * table d'indices (de meme taille que test_images) est allouee et
 * doit etre liberee par l'appelant apres usage. */
uint32_t* brute_force_NN(struct array* dataset, struct array* test_images,
                         image_distance dist_func);

/* recherche du plus proche voisin au sens de la distance "dist_func"
 * pour un groupe d'image tests, grace a un kd-arbre "kdtree". La
 * table d'indices (de meme taille que test_images) est allouee et
 * doit etre liberee par l'appelant apres usage. */
uint32_t* kdtree_approx_NN(struct kdtree*, struct array* test_images,
                           image_distance dist_func);
```

5.2 Modules et implémentation fournis

Module array. Le module `array` définit un allocateur et accesseur de tableau d'images, avec la propriété que toutes les images d'un même tableau ont des dimensions communes données par `get_width` et `get_height`.

```
/* creation d'un tableau de "size" images, toute de taille width x
 * height, chaque pixel faisant 1 octet */
struct array* create_array(uint32_t size, uint32_t width, uint32_t height);

/* changer la taille du tableau d'images */
void resize_array(struct array* array, uint32_t new_size);

/* liberer la memoire du tableau d'images */
void free_array(struct array* array);

/* accesseur/modifieur: renvoie un pointeur sur l'image no "elem",
 * c'est a dire un tableau de pixels adressables de la maniere
 * suivante (uint8_t*)get_elem(a, elem)[y*get_width + x] */
/* precondition: elem est present dans a */
void* get_elem(const struct array* a, uint32_t elem);

/* nombre d'images dans le tableau a */
uint32_t get_size(const struct array* a);

/* largeur des images du tableau */
uint32_t get_width(const struct array* a);

/* hauteur des images du tableau */
uint32_t get_height(const struct array* a);

/* indice "elem" d'une image du tableau a, a partir de son pointeur
 * data (obtenu avec get_elem). */
/* precondition: elem est present dans a */
uint32_t get_index(const struct array* a, const void* data);
```

Module image. Le module `image` fournit des fonctions utilitaires permettant de charger les données de la base MNIST - images et étiquettes (labels). Il permet également le chargement et la sauvegarde de fichiers image au format pgm, utiles pour le débogage et pour écrire les programmes d'application. Enfin il fournit une fonction utilitaire permettant de réduire les dimensions des images d'un tableau d'images (`image_downscale`), pour passer des dimensions MNIST 28×28 à une taille plus réduite, 10×10 par exemple. Les fonctions de chargement `load_*` créeent toutes un tableau d'images ou d'étiquettes (appel `create_array` en interne). Il faudra donc pour chacune bien appeler `free_array` pour libérer l'espace correspondant. Pour le cas des étiquettes il s'agit d'images de taille 1×1, avec un seul entier de type `uint8_t` représentant l'étiquette (valeur du chiffre). Pour la fonction `load_pgm_image` chargeant une image pgm, un tableau d'une seule image est créé avec l'image chargée.

```
/* remttre les images du tableau "a" a l'echelle avec une nouvelle
   taille plus petite width x height */
struct array* image_downscale(const struct array* a,
                              uint32_t width, uint32_t height);

/* charger les images d'un fichier .gz au format MNIST (decrit
 * http://yann.lecun.com/exdb/mnist/) */
struct array* load_image_array(char *gzfilename);

/* charger les etiquettes correspondantes d'un fichier .gz, au format
 * MNIST (http://yann.lecun.com/exdb/mnist/) */
struct array* load_label_array(char *gzfilename);

/* charger un fichier pgm dans un tableau (a une seule entree) */
```

```
struct array* load_pgm_image(char *pgmfilename);

/* sauvegarder l'element d'indice "elem" du tableau au format pgm */
void save_pgm_image(char *filename, const struct array* a, uint32_t elem);

/* definit le type pointeur de fonction de distance entre deux images
 * (pointeur vers tableau d'octet brut tel que stockes pour chaque
 * element d'un tableau array). Necessite donc de passer la taille des
 * images weight x height */
typedef float (*image_distance)(const uint8_t* imga,
                                const uint8_t* imgb,
                                uint32_t width, uint32_t height);

/* une distance euclidienne est fournie entre deux images */
float image_euclidean_distance(const uint8_t* imga, const uint8_t* imgb,
                               uint32_t width, uint32_t height);
```

5.3 Données fournies.

Nos fournissons les fichiers de la base MNIST à l'endroit suivant : `/matieres/3MMPLC/MNIST`. Utiliser un lien symbolique avec `ln -s` sur ce répertoire pour ne pas gaspiller d'espace sur les disques de l'école. La base contient 10′000 images test (`t10k-images-idx3-ubyte.gz`) et 60′000 images de la base d'apprentissage (`train-images-idx3-ubyte.gz`). Toutes deux sont étiquetées, les fichiers d'étiquette sont réciproquement `t10k-labels-idx1-ubyte.gz` et `train-labels-idx1-ubyte.gz`. La base d'apprentissage est étiquetée par définition, celle des tests l'est aussi pour vérification des performances de l'algorithme. On peut pour cela calculer des statistiques sur le taux de succès/erreur de la reconnaissance, connaissant la vraie réponse pour les images de la base de test.

Références

[1] Jon Louis Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4) :214–229, April 1980.

[2] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3) :209–226, September 1977.