Dans ce TP, on souhaite mettre en place des algorithmes d'apprentissage automatique dans l'objectif de faire de la reconnaissance de chiffres manuscrits. Pour ce faire, on utilisera la *MNIST handwritten digit database* (*Modified National Institute of Standards and Technology*), disponible sur le site `http://yann.lecun.com/exdb/mnist/`.

Cette base de données contient des images de chiffres manuscrits étiquetés (de 0 à 9). On trouve sur le site 4 fichiers (qu'on téléchargera et qu'on extraira dans un dossier de travail) :

- un fichier contenant 60000 images de chiffres d'entraînement ;
- un fichier contenant les étiquettes de chacun des chiffres précédents ;
- la même chose pour 10000 images et étiquettes, correspondant à des données de test.

La figure 1 montre un échantillon des images disponibles dans les fichiers.

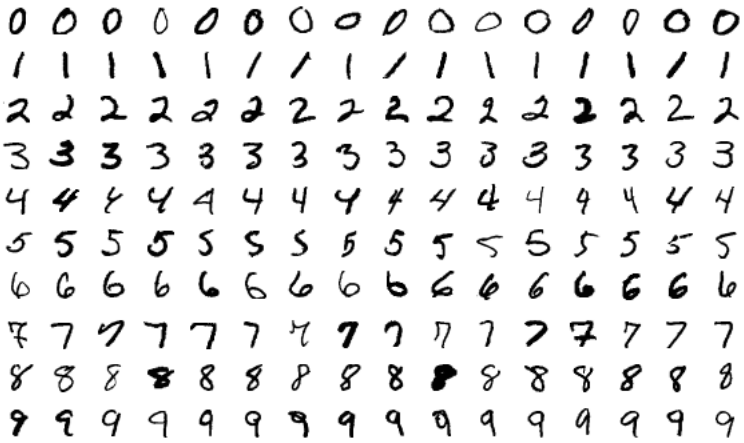

FIGURE 1 – Échantillon d'images de la base MNIST

Par ailleurs, le fichier `TP12-outils.ml` (trouvable sur le site `https://github.com/nathaniel-carre/MPI-LLG`) contient le code nécessaire à la lecture des fichiers précédents. On rappelle qu'on peut le charger dans un autre fichier OCaml par la commande (en remplaçant le nom du chemin) :

```
#use "Chemin/du/fichier/TP12-outils.ml";;
```

On peut également faire de la compilation des deux fichiers (fichier d'outils et fichier de travail), de la forme : `ocamlc TP12-outils.ml TP12.ml`.

La première ligne du fichier est à modifier pour indiquer le chemin du dossier de travail contenant les fichiers précédents. La fonction `ouvrir` et la ligne qui suit permettent de créer deux tableaux, `entrainement` de taille 60000 et `test` de taille 10000, contenant des images étiquetées, une image étant un objet du type suivant (déjà défini dans le fichier) :

```
type image = {matrice : int array array; etiquette : int};;
```

Ainsi, si `im` est de type `image`, alors `im.matrice` est une matrice de taille 28×28 tel que `im.matrice.(i).(j)` est la valeur du niveau de gris (entre 0, noir, et 255, blanc) du pixel ligne i et colonne j de l'image associée, et `im.etiquette` est un entier entre 0 et 9 correspondant au chiffre écrit sur l'image.

On trouve également dans ce fichier `.ml` deux fonctions d'affichage et de lecture d'image (qui seront facultatives pour travailler sur le TP, notamment s'il y a des problèmes d'installation). Des explications se trouvent dans les commentaires.

Exercice 1

1. Écrire une fonction `delta : int array array -> int array array -> int` qui calcule le carré de la distance euclidienne entre deux images. On considèrera une image comme un vec-

teur de $\mathbb{R}^{28\times 28} = \mathbb{R}^{784}$.

Le fichier `TP12-outils.ml` contient une implémentation de file de priorité de type `('a, 'b) tas` disposant des primitives suivantes :

- si `tas` est un élément de type `('a, 'b) tas`, alors `tas.taille` est le nombre d'éléments présents dans le tas ;
- `creer_tas : unit -> ('a, 'b) tas` crée un tas vide ;
- `inserer : 'a -> 'b -> ('a, 'b) tas -> unit` prend en argument une priorité p, un élément x et un tas et insère l'élément x avec priorité p dans le tas ;
- `maximum : ('a, 'b) tas -> 'a * 'b` renvoie le couple (p, x) tel que x est l'élément de priorité p maximale parmi les éléments du tas (mais sans modifier le tas) ;
- `extraire_max : ('a, 'b) tas -> 'a * 'b` extrait le couple (p, x) de priorité maximale du tas et le renvoie ;
- `tas_vers_tab : ('a, 'b) tas -> ('a * 'b) array` renvoie un tableau contenant les couples (priorité, valeur) des éléments du tas, triés par priorité croissante.

2. Écrire une fonction `plus_proches_voisins : int -> image array -> int array array -> int array` qui prend en arguments un entier K, un tableau d'images étiquetées t et une image I et renvoie un tableau de taille K contenant les étiquettes des K éléments de t qui sont les plus proches de I (au sens de la distance euclidienne). On supposera que K sera inférieur à la taille de t.
3. Écrire une fonction `majoritaire : int array -> int` qui prend en argument un tableau d'étiquettes et renvoie l'étiquette majoritaire présente dans ce tableau.
4. En déduire une fonction `knn : int -> image array -> int array array -> int` qui prend en arguments un entier K, un tableau d'images t et une image I et applique l'algorithme KNN (*k-nearest neighbours*) pour déterminer l'étiquette prédite pour I.
5. Écrire une fonction `matrice_confusion : int -> image array -> image array -> int array array` qui prend en argument un entier K, un tableau d'images correspondant à des données d'entraînement et un tableau d'images correspondant à des données de test et crée et renvoie la matrice de confusion des données de test correspondant à l'algorithme KNN avec les données d'entraînement.
6. Écrire de même une fonction `taux_erreur` prenant les mêmes arguments et renvoyant un flottant correspondant au taux d'erreurs.
7. Tester les deux fonctions précédentes avec les données fournies et une valeur de $k = 3$. Étant donnée la taille de la base de données, on utilisera des sous-tableaux (avec `Array.sub`) de taille 2000 et 400 (ou moins) respectivement pour les données d'entraînement et de test.
8. Comparer les résultats obtenus selon la valeur de k. Comparer aux résultats obtenus en utilisant la distance Manhattan.

Exercice 2

On souhaite accélérer le temps de calcul précédent. Pour ce faire, on remarque qu'une image contient une majorité de pixel noirs. Lors d'un calcul de distance, beaucoup de temps est donc passé à calculer des termes $(0-0)^2$. Pour éviter cela, on peut déterminer, pour chaque image, la liste de ses pixels significatifs (contenant de l'information).

1. Écrire une fonction `liste_blancs : int array array -> (int * int) list` qui prend en argument une matrice correspondant à une image en niveau de gris (qu'on pourra supposer de dimensions 28×28) et renvoie une liste de couples d'indices (i, j) telle que la luminosité est supérieure à 10 sur le pixel de coordonnées (i, j) dans l'image.

Pour éviter de recalculer la liste des pixels significatifs, on créera des tableaux de listes correspondant aux listes de pixels significatifs des données d'entraînement et de test.

2. Quelle est la proportion de pixels significatifs moyenne sur les images des données d'entraînement ?

3. Écrire une fonction `delta2 : int array array -> int array array -> (int * int) list -> (int * int) list -> int` qui prend en argument deux images, ainsi que les deux listes contenant leurs pixels significatifs, et renvoie le carré de la distance entre les deux images, en ignorant les pixels non significatifs.
4. Réécrire les fonctions de l'exercice précédent en modifiant ce qui est nécessaire pour prendre en compte ces pixels significatifs.
5. Vérifier que le temps de calcul est effectivement raccourci avec cette nouvelle méthode.

Pour la suite, on souhaite faire du clustering en utilisant la base de données MNIST et vérifier que les classes obtenues contiennent bien des éléments étiquetés de la même manière.

Exercice 3

Dans cet exercice, on souhaite appliquer l'algorithme des k-moyennes. Les valeurs des pixels des barycentres seront arrondies à l'entier le plus proche. On pourra adapter les signatures des fonctions pour utiliser la fonction de distance entre images de l'exercice précédent si nécessaire.

1. Écrire une fonction `echantillon : int -> 'a array -> 'a array` qui prend en argument un entier m et un tableau t de taille N et renvoie un tableau de taille m contenant des éléments distincts de t choisis aléatoirement et uniformément. On interdira toute modification de t et on supposera $N \gg m$.

   On rappelle que `Random.int n` *renvoie un entier choisi aléatoirement et uniformément entre* 0 *et* $n-1$.
2. Écrire une fonction `plus_proche : int array array array -> int array array -> int` qui prend en argument un tableau d'images t (sous forme matricielle), une image I (sous forme matricielle) et renvoie l'indice de l'élément de t qui est le plus proche de I.
3. Écrire une fonction `classes : image array -> int array array array -> int array` qui prend en argument un tableau d'images étiquetées de taille N et un tableau de barycentres de taille m et renvoie un tableau classe de taille N contenant des éléments entre 0 et $m-1$, tel que la classe de l'image d'indice i correspond à l'indice du plus proche barycentre.
4. Écrire une fonction `barycentres : int -> image array -> int array -> int array array array` qui prend en argument un entier m, un tableau d'images étiquetées de taille N et un tableau de classes de taille N et renvoie un tableau de barycentre de taille m, le barycentre d'indice j étant calculé à partir de toutes les images de classe j.
5. En déduire une fonction `k_moyennes : int -> int -> image array -> int array` qui applique l'algorithme des k-moyennes pour attribuer une classe à chaque image. La fonction prendra en argument le nombre de classes, le nombre maximal d'itérations et le tableau d'images.

On souhaite maintenant vérifier la pertinence de cette fonction sur nos données.

6. Déterminer une matrice de confusion de l'algorithme des k-moyennes sur le jeu de test. Les chiffres sont-ils bien différenciés ?

Exercice 4

Implémenter un clustering hiérarchique ascendant pour créer 10 classes. On pourra tester avec différents types de liaisons.