Chaima JAZIRI	Ecole Nationale d'Ingénieurs de Sousse	Année universitaire : 2020-2021

Mini-projet

Module : Programmation orientée objet

Intitulé projet : Reconnaissance de l'écriture manuscrite

1. Présentation

Le but de ce projet est d'implémenter et d'évaluer un programme capable de reconnaître des chiffres écrits à la main. Cette tâche de vision par ordinateur trouve de nombreuses applications pratiques dont la plus importante est sans doute la numérisation de documents. Elle peut facilement être généralisée pour reconnaître des caractères autres que des chiffres.
Pour réaliser notre objectif, nous allons utiliser un algorithme d'apprentissage automatique (machine learning) : au lieu d'expliquer à l'ordinateur comment reconnaître des chiffres à travers de règles (instructions conditionnelles if/else), nous lui montrerons une large collection d'images de chiffres manuscrits en lui indiquant, pour chaque image, à quel chiffre elle correspond. Ceci permettra au programme, par la suite, de traiter et d'interpréter de nouvelles images de chiffres manuscrits.

a. L'algorithme des K plus proches voisins

La discipline de l'apprentissage automatique se sert de connaissances mathématiques et statistiques pour exploiter les immenses bases de données que l'on désigne aujourd'hui sous le vocable de *big data*. L'idée est d'extraire des connaissances (et de « l'intelligence ») non pas en enseignant aux machines des règles fixes mais en leur permettant *d'apprendre* en traitant beaucoup d'exemples.
Le traitement de bases de données si larges impose une programmation très attentive aux performances : la complexité des algorithmes et la mémoire utilisée peuvent faire toute la différence entre un programme qui s'exécute en quelques secondes ou en plusieurs heures.
Parmi les nombreuses applications de cette nouvelle discipline nous allons nous concentrer sur la classification supervisée : le programme recevra une image et devra lui attribuer une étiquette représentant le chiffre écrit sur l'image. L'ensemble des étiquettes est fixe et connu au départ ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9}) et le programme apprend en recevant un ensemble d'images déjà étiquetées.
L'algorithme que nous utiliserons s'appelle **k-nearest neighbors** ou **méthode des k plus proches voisins**, abrégé KNN. Il s'agit d'un algorithme simple mais puissant qui prend en entrée une image à classifier, une valeur *k* et une base d'images déjà classifiées. L'algorithme commence par identifier les *k* images déjà classifiées qui sont les plus *proches* de l'image à classifier. Il choisira ensuite l'étiquette la plus fréquente parmi ces *k* images retenues.

Les deux éléments fondamentaux de cette méthode sont le calcul de la proximité ou *distance* entre l'image à classifier et celles déjà classifiées et l'agrégation permettant de choisir une étiquette dans l'ensemble des images les plus proches.

Cet algorithme permet de s'approcher au monde de l'apprentissage automatique sans avoir besoin de connaissances avancées en analyse et statistiques comme la dérivation des gradients, la théorie de l'échantillonnage, les algorithmes de dérivation automatique et l'algèbre linéaire qui caractérisent des algorithmes plus complexes comme les réseaux neuronaux.

b. La base de données MNIST

L'ingrédient fondamental pour toute recette d'apprentissage automatique est l'ensemble des données qui permettent à l'algorithme d'apprendre. Dans le cadre de ce projet, nous utiliserons la base de données MNIST qui est un sous ensemble d'un « dataset » développé par le National Institute of Standards and Technology (NIST) du gouvernement des États-Unis. Il est divisé en deux parties : une partie destinée à l'apprentissage (training) et une partie pour tester l'efficacité des algorithmes. La séparation de la base de données en plusieurs parties est nécessaire pour évaluer les algorithmes et nous permet de mesurer comment ceux-ci se comportent face à des échantillons qu'ils n'ont pas traités en phase d'apprentissage. Il se peut en effet que l'algorithme apprenne à reconnaître très bien les images destinées à l'apprentissage mais qu'il fasse des erreurs importantes sur des images inconnues (overfitting).

Le sous ensemble destiné à l'apprentissage de MNIST contient 60000 images en noir et blanc ainsi que les étiquettes correspondantes alors que la partie destinée aux tests en contient 10000.

Les images ont été normalisées pour avoir les mêmes dimensions et centrées. Elles sont stockées au format IDX décrit à la section 3. Nous avons quatre partitions de tailles différentes du sous ensemble destiné à l'apprentissage avec 10, 100, 1000 et 5000 images par étiquette ainsi que le dataset de test en entier avec les 10000 images.

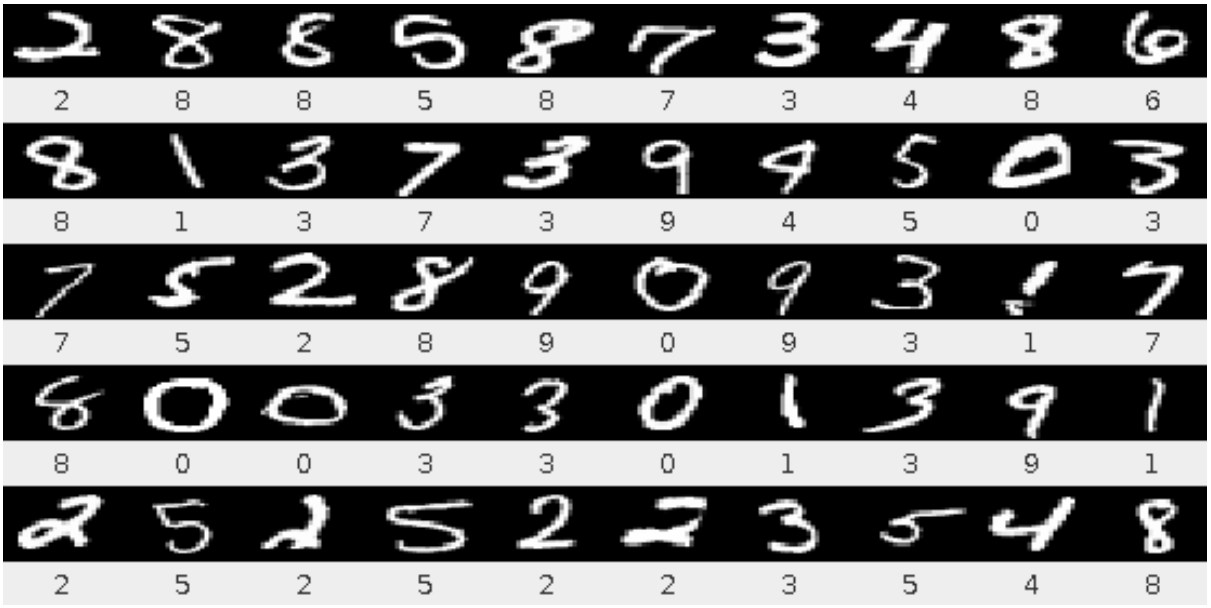

Fig. 1 : Un petit extrait de la base de données MNIST

2. Structure et code fournis

Le projet est articulé en quatre parties :

- La création d'un parseur pour le format IDX utilisé par le dataset MNIST. Le rôle du parseur est d'importer des données enregistrées dans le format IDX dans des structures de données utilisables par un développeur Java.
- L'implémentation de deux fonctions permettant de mesurer la similarité entre une image et les images contenues dans le dataset d'apprentissage.
- L'implémentation d'un algorithme permettant aux K images du dataset les plus proches de l'image à classifier de voter pour leur étiquette et de permettre ainsi le choix de l'étiquette ayant reçu le plus de votes.
- L'implémentation d'outils permettant d'évaluer l'impact du choix de la valeur de K, de la taille du dataset d'apprentissage et de la fonction de distance utilisée sur la précision du programm

Le fichier Helpers.java simplifie l'interaction avec les fichiers binaires et l'affichage d'images :

- *byte[] readBinaryFile(String path)* ouvre le fichier se trouvant dans le chemin, path, fourni et retourne son contenu en bytes.
- Plusieurs variations de la méthode show permettent de visualiser les images, les images et leurs étiquettes, les images et si les étiquettes associées sont correctes. Elles affichent une fenêtre à l'écran (dont le titre est passé en argument) et attendent que l'utilisateur la ferme pour continuer l'exécution :
  - void show(String title, byte[][][] tensor, int rows, int columns) affiche les images contenues dans le tenseur tensor dans une grille de dimension rows × columns. Seules les premières rows × columns images seront lues. Le terme tenseur sera expliqué dans ce qui suit : chaque image sera représentée utilisant un tableau bidimensionnel et un ensemble d'images sera représenté comme un tableau à trois dimensions (un tenseur).

Les tableaux produits par les méthodes de *Helpers.java* sont considérés comme non nuls et correctement constitués.

3. Tâche 1 : Traitement du format IDX

Dans cette première partie, nous avons extrait des images et des étiquettes à partir de fichiers binaires utilisant le format IDX.

Il convient d'abord de comprendre les objectifs visés. Nous avons appris à comparer une image à classifier avec un ensemble d'images déjà classifiées (que nous avons extraites depuis des fichiers binaires).

L'un des problèmes principaux qui se pose alors est de savoir stocker l'ensemble des images déjà classifiées. Il convient d'être très attentif au choix de cette structure de données appelée à être extrêmement volumineuse. En effet, pour pouvoir traiter des bases de données de taille importante, l'écriture du code impose une attention particulière aux performances, en particulier à la complexité des algorithmes et à l'utilisation de la mémoire.

L'idée est de représenter l'ensemble des images classifiées comme un tenseur à 3 dimensions (voir le cas le plus à droite de la figure 2).

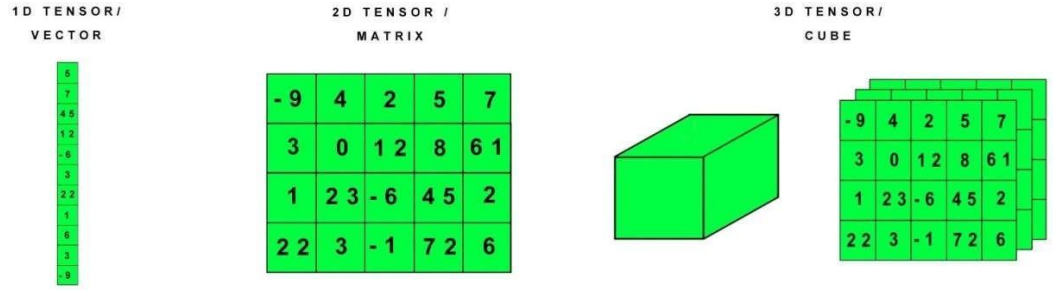

Fig. 2 : Un tenseur à trois dimensions n'est rien d'autre qu'un tableau contenant des tableaux à deux dimensions

Concrètement, il s'agira donc d'une pile d'images, telle que schématisée par la figure 3.

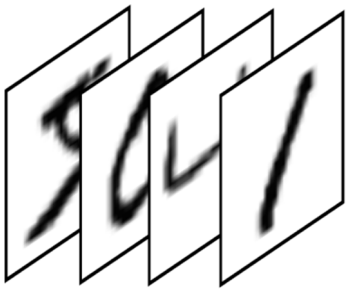

Fig. 3 : Le format utilisé pour stocker les images dans votre programme ressemble donc à une pile d'images

Une image est quant à elle simplement une matrice (tableau à deux dimensions) de pixels. Enfin, un pixel est modélisé au moyen de sa couleur.
Nous travaillerons avec des images en niveau de gris où un pixel (plus précisément sa couleur) peut être modélisé par un entier entre 0 et 255.
Comme un des objectifs est de minimiser la taille des tenseurs, il convient de choisir le type le plus économe pour modéliser la couleur d'un pixel et nous travaillerons pour cela avec le type byte.
Dans cette partie du projet, nous allons commencer par définir précisément la mise en œuvre des tenseurs (et également des ensembles d'étiquettes). Nous implémentons ensuite le « parsing » de fichiers IDX qui nous permettra de construire les tenseurs en important des images de la base de données MNIST. Nous expliquerons aussi cette base de données pour extraire les ensembles d'étiquettes.

a. Le format de destination

Le dataset que nous allons exploiter est composé de paires de fichiers binaires contenant respectivement les étiquettes et les images.
La correspondance entre images et étiquettes se fait par indice : le chiffre écrit sur l'image à la position *i* est donné par l'étiquette enregistrée à la position *i*.

Notre tâche consistera à parser ces fichiers pour construire des tableaux d'étiquettes et des tenseurs d'images exploitables dans le cadre du projet.

Les étiquettes représentant les chiffres auront une valeur entière comprise entre 0 et 9. Nous utiliserons naturellement aussi le type byte pour modéliser ces entiers de petite taille.

Nous stockerons donc les valeurs des étiquettes dans des tableaux de tailles fixes byte[].
Nos tenseurs d'images seront comme suggéré plus haut, des tableaux de taille fixe à trois dimensions byte[][][]stockant des (couches de) pixels.

Dans le format IDX, les valeurs des pixels des images sont des valeurs entières codées sur 8 bits des nombres signés, ceci va donc nécessiter quelques précautions lors de la construction des tenseurs.

b. Le format IDX

Cette section décrit le format des fichiers binaires contenant les étiquettes et les images.

Le format des fichiers contenant les étiquettes

Les 4 premiers bytes d'un fichier forment un entier appelé nombre magique (magic number) qui, pour les fichiers d'étiquettes, est égal à 2049. Les 4 bytes suivants forment un autre entier déterminant le nombre d'étiquettes dans le fichier. Chaque byte qui suit est alors une étiquette.

Le format des fichiers contenant les images

Les 4 premiers bytes d'un fichier forment aussi le nombre magique qui, pour les fichiers d'images, est égal à 2051. Les 4 bytes suivants forment un entier déterminant le nombre d'images. Ce nombre est suivi par 4 bytes qui forment un entier indiquant la hauteur des images suivis par 4 bytes donnant la largeur des images. Après ces 4 entiers, chaque byte est la valeur du niveau de gris d'un pixel de l'image. -128 indique un pixel noir alors que 127 indique un pixel blanc. Les valeurs des pixels sont enregistrées consécutivement, une ligne après l'autre : la première valeur correspond au pixel haut-gauche et la dernière au pixel bas-droite.

Chaque image sera enregistrée dans un tableau de taille fixe à deux dimensions byte[][] dont la première dimension représente la hauteur et la deuxième dimension la largeur. Si on appelle image ce tableau, alors image[0][0] correspond au pixel haut-gauche de l'image. Les images (qui auront toutes les mêmes dimensions) seront empilées dans un tenseur de type byte[][][]. Ainsi, pour une variable d'exemple byte[][][] tensor on aura que :

- *tensor.length* est le nombre d'images
- *tensor[0].length* est la hauteur des images (le nombre de lignes)
- *tensor[0][0].length est* la largeur des images (le nombre de colonnes)

c. Parsing de fichiers binaires au format IDX

Notre première tâche est donc de compléter les deux méthodes suivantes qui transforment un tableau de bytes respectivement en un tableau d'étiquettes et en un tenseur d'images.
Nous implémentons la méthode utilitaire

```
public static int extractInt( byte b31ToB24 , byte b23ToB16, byte b15ToB8, byte b7ToB0)
```

qui assemble 4 bytes en un *int* puis les deux méthodes

```
public static byte[][][] parseIDXimages( byte[] data) public static
  byte[] parseIDXlabels( byte[] data)
```

qui réalisent respectivement :

- l'extraction d'un tenseur d'images à partir d'un fichier binaire *data*.
- l'extraction d'un ensemble d'étiquettes à partir d'un fichier binaire *data*.

Nous nous aiderons de la méthode ***extractInt*** pour coder les traitements requis.
Le programme principal fourni dans le fichier KNN.java nous montre comment tester nos méthodes au fur et à mesure.

Exemple de code qui lit le dataset IDX

```
  // Charge les étiquettes depuis le disque byte[] labelsRaw =
        Helpers.readBinaryFile("10-per-digit_labels_train");
// Parse les  étiquettes
byte[] labelsTrain = parseIDXlabels( labelsRaw);
// Affiche le nombre de labels
  System.out.println(labels.length);
// Affiche le premier label System.out.println(labels[0]);
  // Charge les images depuis le disque byte[] imagesRaw =
        Helpers.readBinaryFile("10-per-digit_images_train");
// Parse les  images
byte[][][] imagesTrain = parseIDXimages( imagesRaw);
  // Affiche les dimensions des images System.out.println("Number
    of images : " + images.length); System.out.println("height : " +
    images[0].length); System.out.println("width : " +
    images[0][0].length);
// Affiche les 30 premières images et leurs étiquettes Helpers. show(" Test",
  imagesTrain, labelsTrain, 2, 15);
```

4. Tâche 2 : Distance entre images

Il s'agit maintenant d'implémenter deux fonctions permettant de mesurer la distance entre deux images :

- une méthode simple permettant de ne parcourir qu'une seule fois les deux images à comparer.
- une seconde, plus complexe, qui nécessitera deux passes sur les deux images.

Nous étudierons, dans la dernière partie du projet, l'impact du choix entre les deux sur la précision du programme.

a. Distance Euclidienne

La première méthode calcule la *distance euclidienne* : on note A la première image, B la deuxième image, H leur hauteur et W leur largeur. La distance euclidienne entre A et B, notée $E(A, B)$ est donnée par:

$$E(A,B) = \sqrt{\sum_{i=0}^{H-1}\sum_{j=0}^{W-1}[A(i,j) - B(i,j)]^2}$$

où $I(i, j)$ indique la valeur du pixel à la ligne i et à la colonne j de l'image I.

Dans le cadre de ce projet les distances sont calculées dans le but d'être comparées entre elles. Ainsi nous pouvons éviter le calcul de la racine carrée :

$$\hat{E} = E(A,B)^2 = \sum_{i=0}^{H-1}\sum_{j=0}^{W-1}[A(i,j) - B(i,j)]^2$$

Les images les plus proches d'une image donnée seront donc celles *minimisant* la distance à cette image.

b. Similarité inversée

La deuxième mesure de distance est obtenue à partir de la corrélation entre deux échantillons. Cette grandeur est souvent utilisée dans le monde de la vision par ordinateur. Il existe dans ce cadre une formule permettant de calculer la corrélation empirique, p(A, B), entre deux images A et B. Cette formule produit des valeurs normalisées dans l'intervalle [−1, 1] et c'est la valeur maximale (1) qui indique une plus grande similarité. Pour pouvoir continuer à minimiser les valeurs permettant de comparer deux images, nous allons plutôt travailler avec la notion de similarité inversée, définie comme étant SI(A, B) = 1 − p(A, B) où p indique la corrélation empirique normalisée évoquée plus haut :

$$SI(A,B) = 1 - \frac{\sum_{i=0}^{H-1}\sum_{j=0}^{W-1}(A(i,j)-\bar{A})(B(i,j)-\bar{B})}{\sqrt{\left(\sum_{i=0}^{H-1}\sum_{j=0}^{W-1}[A(i,j)-\bar{A}]^2\right)\left(\sum_{i=0}^{H-1}\sum_{j=0}^{W-1}[B(i,j)-\bar{B}]^2\right)}}$$

où $\bar{I}$ indique la moyenne des valeurs des pixels de l'image I :

$$\bar{I} = (H \times W)^{-1} \sum_{i=0}^{H-1}\sum_{j=0}^{W-1} I(i,j)$$

On remarque que SI(A, B) ∈ [0, 2] où 0 indique que A et B sont égales et 2 indique que les deux images sont complètement différentes.

Pour cela nous aurons besoin d'une méthode pour calculer la moyenne d'une image. Pour accélérer l'exécution du programme considéré la possibilité de calculer la moyenne des deux images en une seule passe avec une méthode qui itère sur les deux images et retourne les deux similarités inversées dans un tableau de taille fixe de longueur 2.

5. Tâche 3 : La méthode des K plus proches voisins

a. Aggregations des distances et choix de l'étiquette

Les deux méthodes implémentées lors de l'étape précédente permettent de calculer la distance entre l'image à classifier et toutes les images de la base d'images déjà classifiées.

Pour choisir les k images les plus proches de l'image à classifier, nous allons trier les indices des images (et donc des étiquettes) selon la valeur croissante de la distance. Ensuite nous pourrons choisir l'étiquette la plus répandue parmi celles des k voisins de l'image à classifier.

Quicksort

Pour trier les indexes des images nous avons choisi un algorithme très utilisé appelé *quicksort* ou tri rapide.

L'algorithme reçoit le tableau à trier ainsi que deux indices low et high qui indiquent l'intervalle du tableau à trier. Par exemple, pour trier le tableau entier, low = 0 et high =tableau.length - 1. Un élément du tableau est choisi comme pivot : tous les éléments plus petits que le pivot sont déplacés à sa gauche alors que les éléments plus grands à sa droite. Finalement l'algorithme s'invoque lui-même (appels récursifs) deux fois : une fois sur la partie à gauche et une fois sur la partie à droite du pivot (donc avec le même tableau mais en modifiant les paramètres low et high).

Algorithm 1 Quicksort

Require : $array$, low, $high$
  $l \leftarrow low$
  $h \leftarrow high$
  $pivot \leftarrow$ élément du tableau à la position low
  while $l \leq h$ **do**
    if $array[l] < pivot$ **then**
      incrémente l
    else if $array[h] > pivot$ **then**
      décrémente h
    else
      Échange les éléments à la position l et h de $array$
      Incrémente l
      Décrémente h
    end if
  end while
  if $low < h$ **then**
    Appelle `quicksort`$(array, low, h)$
  end if
  if $high > l$ **then**
    Appelle `quicksort`$(array, l, high)$
  end if

Étant donné que nous sommes intéressés au tri **des indices** des images, nous apporterons les modifications suivantes à l'algorithme :

- Il recevra un argument en plus : un tableau d'entiers contenant les indices des images, c'est-à-dire qu'au premier appel de la méthode il contiendra 0, 1, 2, 3, . . .
- Au moment d'échanger les éléments l, h d'array, il échangera aussi les éléments l, h du tableau d'indices introduit au point précédent.

La méthode utilitaire échange les éléments aux positions i, j des deux tableaux.

```
public static void swap(int i, int j, float[] values , int[] indices)
```

Choix de l'étiquette

Après avoir trié les indices des images selon la distance de l'image à classifier, les *k* images les plus proches doivent voter pour leur étiquette et l'étiquette avec le plus de votes sera choisie pour l'image à classifier. Dans un premier temps nous réaliserons le décompte des votes des *k* images les plus proches, puis nous chercherons l'étiquette la plus populaire. Si plusieurs étiquettes ont le même nombre de votes, nous choisirons la première dans l'ordre.

b. Construction du classificateur

Toutes les méthodes nécessaires pour construire le classificateur de chiffres manuscrits sont désormais en place. Nous complétons donc le corps de la méthode ***knnClassify*** qui construit le tableau des distances entre l'image à classifier et les images déjà classifiées à l'aide d'une des deux méthodes de distance implémentées lors de l'étape 4. Ensuite, nous utilisons le tableau d'indices triés généré au moyen de ***quicksortIndices*** afin d'extraire l'étiquette retenue pour le chiffre à classifier par le biais de la méthode ***electLabel***.

1. Tâche 4 : Évaluation

Il est impossible d'évaluer les performances du classificateur implémenté jusqu'ici en observant uniquement la fenêtre produite par show, et nous nous poserons sans doute des questions telles que : *Quelle combinaison de K et de taille de dataset d'apprentissage donnent la meilleure précision?*
Dans cette partie nous implémentons quelques moyens simples d'étudier les performances de notre modèle d'apprentissage automatique en considérant la **précision** comme critère.

Une simple mesure de la performance peut être obtenue en calculant le rapport entre le nombre d'étiquettes correctement prédites et le nombre total de prédictions. Ce calcul produit une valeur entre 0 et 1 qui est souvent multipliée par 100 et considérée comme un pourcentage. On définit donc la précision (accuracy) :

$$a = n^{-1} \sum_{i}^{n} \mathbb{1}\{p_i = e_i\}$$

où pi est le ième élément du vecteur d'étiquettes prédites, ei est le i-ème élément du vecteur des vrais étiquettes et n est le nombre de prédictions. 1 est appelé fonction indicatrice : 1{pi = ei} est égal à 1 si pi = ei et 0 dans le cas contraire.

1. Références

- Algorithmes des K plus proches voisins : https://en.wikipedia.org/wiki/K-nearest_ neighbors_algorithm
- Tri rapide : https://fr.wikipedia.org/wiki/Tri_rapide
- Similarité(Pearsoncorrelationcoefficient):https://en.wikipedia.org/wiki/Pearson_ correlation_coefficient